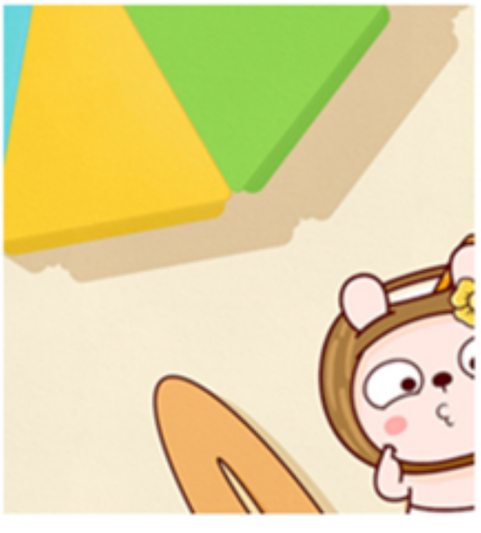
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

严禁网络传播本PDF，违者责任自负！

前微软高级技术布道者撰写，Solidity初学者指南

既涵盖Solidity编程的核心概念，又详细讲解智能合约开发、部署、

[PACKT]

区块链技术丛书

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

SOLIDITY PROGRAMMING ESSENTIALS

Solidity编程

版权所有，严禁传播，违者自负法律责任！

从零开始构建区块链智能合约的初学者指南

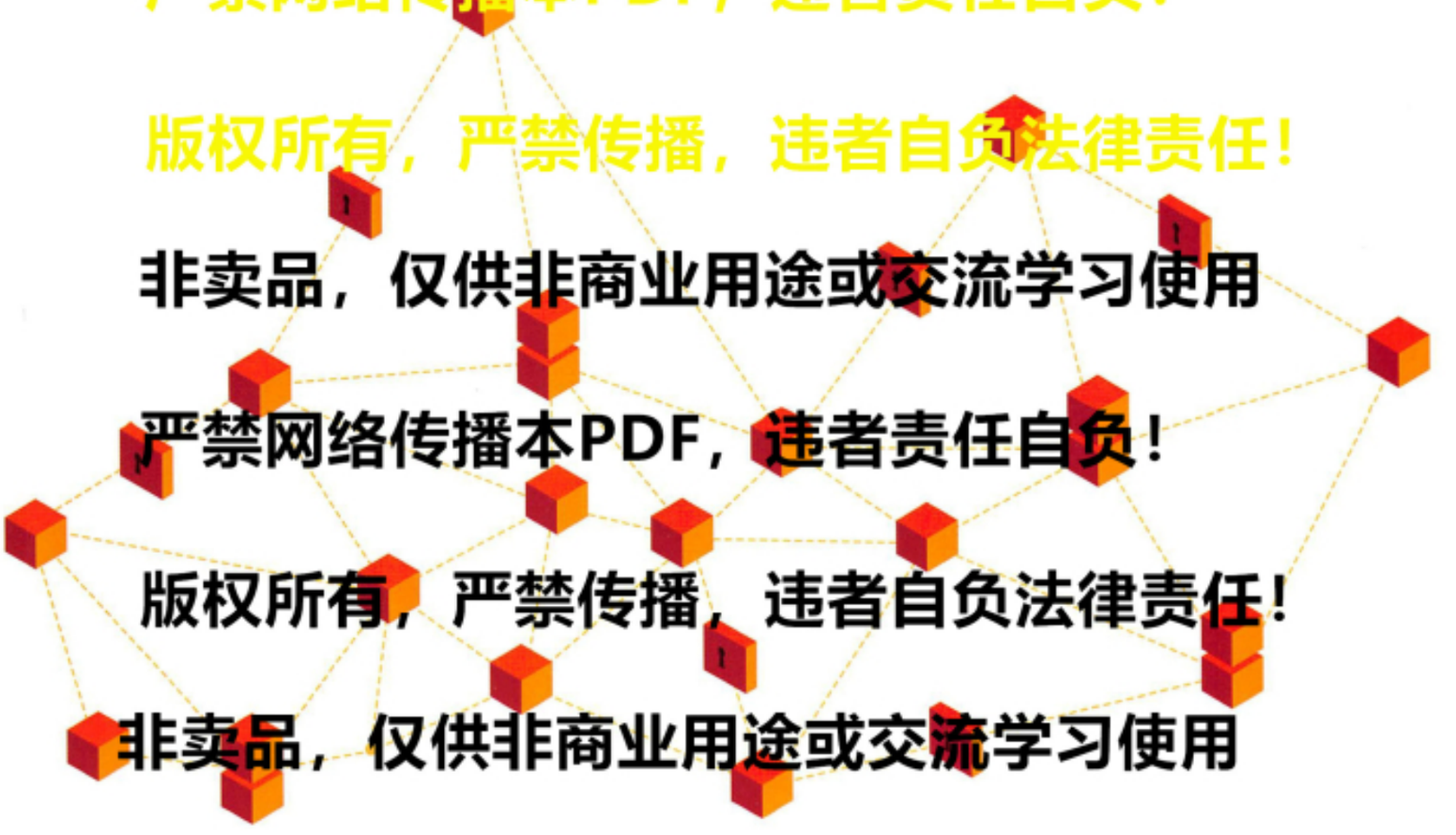
非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

[印] 瑞提什·莫迪（Ritesh Modi）◎著

[illegible] ◎译



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

机械工业出版社
China Machine Press

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

作者简介

瑞提什·莫迪（Ritesh Modi）

前微软高级技术布道者以及地区负责人。他具有丰富的以太坊和 Solidity 开发经验，是区块链及 Solidity 很多相关会议的演讲嘉宾。他是架构师、布道者、演讲者以及著名的领导者，在区块链、数据中心、Azure Bots、认知服务、DevOps、人工智能以及自动化领域都做出了贡献。他也是 5 本书的作者。

译者简介

毛明旺

某互联网公司人工智能方向高级工程师，之前曾在某知名通讯公司担任研发经理、系统工程师、PMP、Scrum Master。从事软件开发多年，对区块链技术兴趣浓厚，是 GitHub 上 Solidity 中文文档的翻译贡献者。

陈冬林

14 余年软件研发与团队管理经验，早期比特币参与者，区块链技术布道者，精通以太坊，具有丰富的智能合约、DApp 与 公链研发经验。现任某金融投资集团区块链总监，同时也是初链核心社区开发者。

林海龙

具有 17 年的 IT 行业经验，长期从事大型企业信息化建设，精通企业架构、大数据、软件开发项目管理。从 2016 年开始研究区块链技术，是去中心化思想的信仰者和传播者。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

华章IT

HZBOOKS | Information Technology

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

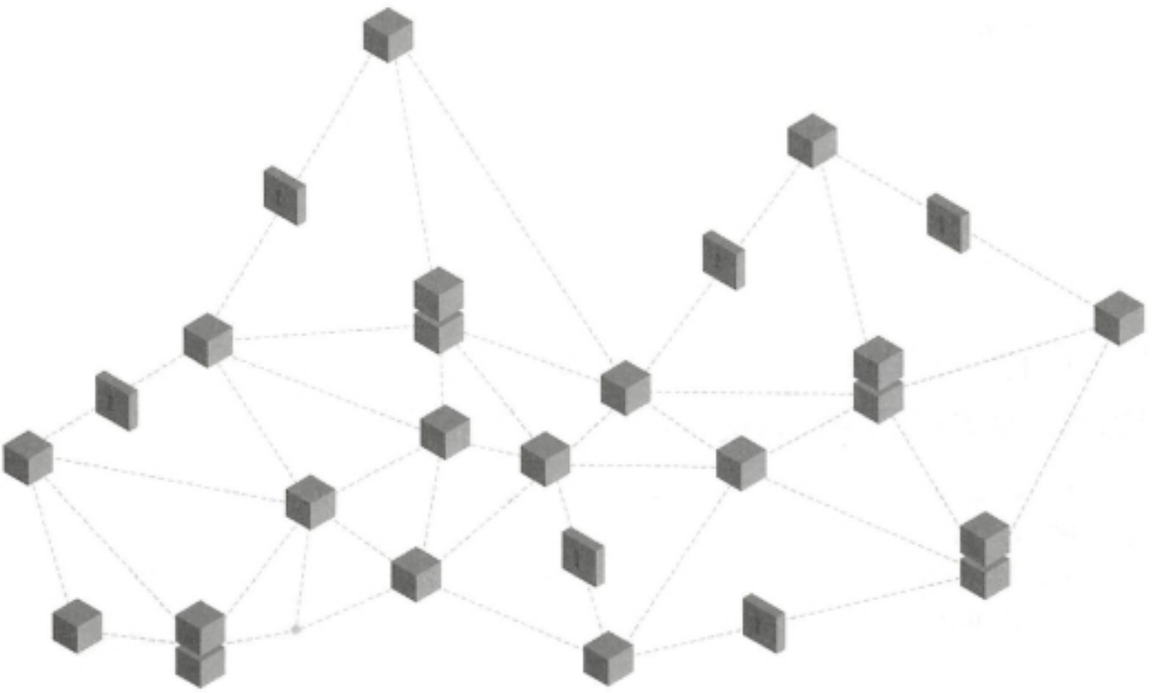
区块链技术丛书

SOLIDITY
PROGRAMMING ESSENTIALS

Solidity编程

构建以太坊和区块链智能合约的初学者指南

[印]瑞提什·莫迪（Ritesh Modi）◎著
毛明旺 林海龙 陈冬林 ◎译



机械工业出版社
China Machine Press

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

图书在版编目（CIP）数据

Solidity 编程：构建以太坊和区块链智能合约的初学者指南 /（印）瑞提什·莫迪（Ritesh Modi）著；毛明旺，林海龙，陈冬林译 . —北京：机械工业出版社，2019.1

（区块链技术丛书）
书名原文：Solidity Programming Essentials

ISBN 978-7-111-61600-9

I. S… II. ①瑞… ②毛… ③林… ④陈… III. 电子商务－支付方式－程序设计
IV. ①F713.361.3 ②TP311.1

中国版本图书馆 CIP 数据核字（2018）第 287840 号

本书版权登记号：图字 01-2018-6833

Ritesh Modi: Solidity Programming Essentials (ISBN: 9781788831383).

Copyright © 2018 Packt Publishing. First published in the English language under the title "Solidity Programming Essentials".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2019 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

Solidity 编程

构建以太坊和区块链智能合约的初学者指南

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：刘 锋 责任校对：殷 虹
印 刷：北京市荣盛彩色印刷有限公司 版 次：2019 年 1 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：12.5
书 号：ISBN 978-7-111-61600-9 定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

严禁网络传播本PDF，违者责任自负！

The Translator's Words 译 者 序

本书从各个方面介绍了如何构建以太坊的智能合约，书中首先介绍了区块链、以太坊和智能合约的核心概念，比如密码学、以太币、gas、区块链和以太坊的架构、以太坊节点、以太坊账户、交易、区块、合约等，随后介绍了安装以太坊和Solidity的一系列工具和环境。

本书各章的主题相对独立，方便时间有限的读者直接钻研感兴趣的章节。对于初学者，我们建议通读本书，在对以太坊、智能合约和Solidity有了一定了解以后，再对某些章节进行详细研究。

Solidity 是以太坊官方推荐的智能合约高级编程语言，这门语言受到了 C++、Python 和 JavaScript 语言的影响，设计的目的是使程序能在以太坊虚拟机（EVM）上运行。

本书一方面从概念上介绍了 Solidity 编程语言，比如从以太坊虚拟机、合约、Solidity 语法和数据结构等方面进行了阐述，另一方面从编写智能合约的角度进行了阐述，比如创建合约、合约组合、封装、继承、多态、方法覆盖、抽象合约、接口、函数、修改器、fallback 函数、异常、事件等诸多方面。最后，介绍了智能合约的测试和调试工具。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

本书的几位译者完全是凭借着对区块链和以太坊的热爱以及在中文社区推广以太坊智能合约开发的一腔热情在业余时间完成了本书的翻译工作。由于自身知识和能力所限，翻译中出现不足甚至谬误在所难免，欢迎各位读者批评指正。

希望这本书能够帮助读者通过学习以太坊 Solidity，学习智能合约开发，进入区块链的技术世界！

译者

2018 年 10 月

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

Preface 前　言

我不太确定上次大量听说关于政府、组织、社区和个人的技术的讨论是什么时候。区块链是一种正在世界各地各种组织中进行详细讨论和辩论的技术。区块链不仅仅是一种有限影响我们生活的技术，而是对我们的生活产生了广泛的影响。不远的将来，区块链将会触及我们生活的方方面面——支付账单，与任何组织进行交易，获得工资，身份认证，教育结果，活动等。这只是开始，我们刚开始了解去中心化的含义及其影响。

我已经在区块链领域工作了很长一段时间，并且一直是加密货币投资者。我是一名技术专家，对比特币非常着迷，因为它架构奇特。我从未遇到过这样优越的思维过程和架构，它不仅解决了经济和社会问题，而且解决了一些技术上未解决的问题，如拜占庭式的一般问题和容错。它在很大程度上解决了分布式计算的问题。

以太坊以几乎相似的方式搭建，当第一次听到并经历智能合约时，我很敬畏。智能合约是在区块链上部署去中心化应用程序并通过自定义逻辑、策略和规则轻松扩展的最大创新之一。

在编写这本书时我心怀愉悦，并真诚地希望你也喜欢阅读和实施 Solidity。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

我介绍了很多我的 Solidity 经验，并尽我所能阐述清楚，希望这本书能让你成为更好的 Solidity 开发人员和优秀的程序员。

如果有哪些我可以做的，能够更好地改善你对这本书的体验，我洗耳恭听！

本书目的读者

为了更好地阅读本书的内容，需要计算和编程的基本概念和知识。如果你觉得自己没有这方面的知识，可以通过快速阅读针对初学者的编程书籍来满足基本要求。本书主要面向使用区块链为最终客户和雇主提供高级服务的区块链架构师、开发人员、顾问和 IT 工程师。如果你想在以太坊上编写智能合约解决方案，那么本书对你来说非常理想。如果你已经拥有一些 JavaScript 经验，那么可以帮助你加快学习速度。

本书主要内容

第 1 章介绍区块链的基本原理、术语和行话、优势、尝试解决的问题以及行业相关性。本章将详细解释重要的概念和架构，还将介绍以太坊特有的概念。在本章中，将讨论有关其外部拥有的账户、合约账户、gas 和以太币等概念。以太坊主要基于密码学，你将了解用于创建交易和账户的散列、加解密算法。本章还将详细解释如何创建交易和账户，如何为每笔交易支付 gas，消息调用和交易之间的差异，以及代码存储和状态管理细节。

第 2 章将指导你使用以太坊平台创建私有区块链。以太坊生态系统中的另

一个重要工具是 ganache-cli。本章还将介绍如何安装 ganache-cli 并使用它来部署 Solidity 合约，如何安装 Solidity 并使用它来编译 Solidity 合约。你还将安装 Mist，

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

这是一个钱包，可以与私有链进行交互。Mist 用于创建新账户，部署并使用合约。本章也将介绍交易中的挖矿。Remix 是创建 Solidity 合约的绝佳工具。

第 3 章开始 Solidity 之旅。在本章中，你将通过了解不同版本以及如何使用预编译指令来学习 Solidity 基础知识。本章的另一个重要方面是如何构建智能合约，将从使用重要结构（如状态变量、函数、常量函数、事件、修改器、fallback、枚举和结构体）方面深入讨论智能合约布局。本章讨论并实现任何编程语言中最重要的元素——数据类型和变量，既有简单数据类型也有复杂数据类型，既有值类型也有引用类型，既有存储类型也有内存类型，我们会使用示例说明所有这些变量类型。

第 4 章提供与区块和交易相关的全局函数和变量、与地址和合约相关的全局函数和变量的实现及使用细节。在一系列智能合约开发时，可以非常方便地使用这些内容。

第 5 章教你如何使用 if ... else 和 switch 语句编写具有条件逻辑的合约和函数。循环是任何语言的重要组成部分，Solidity 提供了 while 和 for 循环用于遍历数组。本章给出了循环的示例和实现。必须根据一定的条件中断循环，并根据其他条件继续循环。

第 6 章是本书的核心章节。从这章开始，你将编写真正的智能合约。本章将会讨论如何编写智能合约，如何定义和实现合约，如何使用 new 关键字和已知地址等不同机制创建和部署合约。Solidity 提供了丰富的面向对象机制，本章将深入研究面向对象的概念和实现，如继承、多继承、声明抽象类和接口，以及为抽象函数和接口提供实现方法。

第 7 章展示了如何实现接受输入和返回输出的基本函数，以及仅输出现有状态而不更改状态和修改器的函数。修改器有助于在 Solidity 中更好地组织代码，

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

并有助于提高合约的安全性和重用合约中的代码。fallback 函数是重要的结构，在函数调用与任何现有函数签名不匹配时执行。对于将以太币发送到合约的情况，实现 fallback 函数也很重要。为了便于理解，将使用示例讨论和实现修改器和 fallback 函数。

第 8 章从合约开发的角度来看是非常重要的。如果出现错误和异常，应将以太币返回给调用者。本章将使用较新的 Solidity 结构（如 assert、require 和 revert）深入解释和实现异常处理，还将讨论 hrow 语句。事件和日志记录有助于理解合约

和函数的执行，本章将展示和解释事件和日志的实现。

第 9 章涵盖了 Truffle 的基础知识，你将理解它的概念，创建一个项目并理解项目结构，修改它的配置，通过一个示例了解编写、测试、部署和迁移合约的整个生命周期。测试合约和编写合约一样重要。Truffle 提供了测试框架，但还应该编写测试用例。本章将讨论单元测试的基础知识，使用 Solidity 编写单元测试，并对智能合约执行单元测试。通过创建交易并验证其结果来执行单元测试，本章将展示编写和执行单元测试的细节。

第 10 章将使用 Remix 和事件等多种工具展示如何解决和调试故障。本章将介绍如何逐行执行代码，检查每行代码执行后的状态，并相应地更改合约代码。

阅读本书的前提条件

本书假定你具有编程的基本知识，理想情况下应具备任意脚本语言的背景。本书的大部分内容需要连接互联网和使用浏览器。有些部分需要一台机器来部署区块链特定的工具和实用程序，可以使用云端或本地部署的物理机器或虚拟机器。

非卖品，仅供非商业用途或交流学习使用

下载示例代码

本书的示例代码可以从 http://www.packtpub.com 通过个人账号下载，也可以访问华章图书官网 http://www.hzbook.com，通过注册并登录个人账号下载。

非卖品，仅供非商业用途或交流学习使用

审校者简介 *About the reviewer*

Pablo Ruiz 在过去的十多年中始终接触前沿技术，参与了数十款技术产品的创造。2008 年，他深入参与了移动游戏和应用的创造；之后，他作为顾问在数字空间领域参与了很多项目。2015 ～ 2016 年，他曾担任拉美地区最著名的风险投

资基金之一的总监，并为公司从零开始创建了自有的金融科技生态系统。2018 年，在活跃参与了数个 ICO 项目之后，他加入 Polymath 成为副总裁并领导研发了第一个基于以太坊的符合监管要求的证券代币发行平台。

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负

5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF

6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅

7、请于下载PDF后24小时内研究使用并删掉本PDF

Contents 目　录

译者序
前言
审校者简介

第1章　区块链、以太坊和智能合约 …… 1

1.1　什么是区块链 …… 2
1.2　为什么是区块链 …… 3
1.3　加密技术 …… 4
1.3.1　散列 …… 5
1.3.2　数字签名 …… 7
1.4　以太币 …… 7
1.5　gas …… 8
1.6　区块链和以太坊架构 …… 9
1.6.1　区块如何互相连接 …… 10
1.6.2　交易和区块如何互相连接 …… 11
1.7　以太坊节点 …… 12
1.7.1　EVM …… 12
1.7.2　以太坊挖矿节点 …… 13
1.7.3　如何挖矿 …… 14
1.8　以太坊账户 …… 15
1.8.1　外部账户 …… 16
1.8.2　合约账户 …… 16
1.9　交易 …… 16
1.10　区块 …… 20
1.11　端到端的交易 …… 21
1.12　什么是合约 …… 22
1.13　什么是智能合约 …… 22
1.14　如何部署合约 …… 27
1.15　本章小结 …… 27

第2章　安装以太坊和Solidity …… 29

2.1　以太坊网络 …… 29
2.1.1　主网 …… 30
2.1.2　测试网络 …… 30
2.1.3　私有网络 …… 31
2.1.4　联盟网络 …… 31
2.2　Geth …… 31
2.3　搭建一个私有网络 …… 35
2.4　ganache-cli …… 40
2.5　Solidity 编译器 …… 43

非卖品，仅供非商业用途或交流学习使用

2.6 web3 JavaScript 库 ····· 43
2.7 Mist 钱包 ····· 45
2.8 MetaMask ····· 47
2.9 本章小结 ····· 51

第3章 Solidity介绍 ····· 53

3.1 以太坊虚拟机 ····· 53
3.2 Solidity 和 Solidity 文件 ····· 54
3.2.1 预编译指令 ····· 55
3.2.2 注释 ····· 56
3.2.3 import 语句 ····· 57
3.2.4 合约 ····· 58
3.3 合约的结构 ····· 59
3.3.1 状态变量 ····· 60
3.3.2 结构 ····· 62
3.3.3 修改器 ····· 63
3.3.4 事件 ····· 64
3.3.5 枚举 ····· 65
3.3.6 函数 ····· 66
3.4 Solidity 中的数据类型 ····· 68
3.4.1 值类型 ····· 68
3.4.2 引用类型 ····· 69
3.5 存储和内存数据位置 ····· 71
3.5.1 规则 1 ····· 71
3.5.2 规则 2 ····· 71
3.5.3 规则 3 ····· 72
3.5.4 规则 4 ····· 72
3.5.5 规则 5 ····· 72
3.5.6 规则 6 ····· 74
3.5.7 规则 7 ····· 75
3.5.8 规则 8 ····· 76
3.6 字面量 ····· 77
3.7 整型 ····· 78
3.8 布尔型 ····· 79
3.9 字节数据类型 ····· 80
3.10 数组 ····· 82
3.10.1 固定数组 ····· 83
3.10.2 动态数组 ····· 83
3.10.3 特殊数组 ····· 84
3.10.4 数组属性 ····· 86
3.11 数组的结构 ····· 86
3.12 枚举 ····· 88
3.13 地址 ····· 89
3.14 映射 ····· 90
3.15 本章小结 ····· 94

第4章 全局变量和函数 ····· 97

4.1 var 类型变量 ····· 97
4.2 变量声明提前 ····· 99
4.3 变量作用域 ····· 100
4.4 类型转换 ····· 101
4.4.1 隐式转换 ····· 102
4.4.2 显式转换 ····· 102
4.5 区块和交易全局变量 ····· 104
4.5.1 交易和消息全局变量 ····· 105
4.5.2 tx.origin 和 msg.sender 的区别 ····· 105

非卖品，仅供非商业用途或交流学习使用

4.6 加密全局变量 …… 106
4.7 地址全局变量 …… 107
4.8 合约全局变量 …… 107
4.9 本章小结 …… 108

第5章 表达式和控制结构 …… 109

5.1 Solidity 表达式 …… 109
5.2 if 决策控制 …… 111
5.3 while 循环 …… 113
5.4 for 循环 …… 114
5.5 do...while 循环 …… 115
5.6 break 语句 …… 116
5.7 continue 语句 …… 117
5.8 return 语句 …… 118
5.9 本章小结 …… 119

第6章 编写智能合约 …… 121

6.1 智能合约 …… 121
6.2 编写一个简单的合约 …… 122
6.3 创建合约 …… 123
6.3.1 使用 new 关键字 …… 123
6.3.2 使用合约地址 …… 124
6.4 构造函数 …… 125
6.5 合约组合 …… 126
6.6 继承 …… 127
6.6.1 单继承 …… 127
6.6.2 多级继承 …… 129
6.6.3 分层继承 …… 130
6.6.4 多重继承 …… 130
6.7 封装 …… 132
6.8 多态性 …… 132
6.8.1 函数多态性 …… 133
6.8.2 合约多态性 …… 133
6.9 方法覆盖 …… 135
6.10 抽象合约 …… 136
6.11 接口 …… 137
6.12 本章小结 …… 139

第7章 函数、修改器和fallback 函数 …… 141

7.1 函数输入和输出 …… 141
7.2 修改器 …… 143
7.3 view 函数、constant 函数和 pure 函数 …… 146
7.4 地址相关函数 …… 148
7.4.1 send 方法 …… 149
7.4.2 transfer 方法 …… 151
7.4.3 call 方法 …… 151
7.4.4 callcode 方法 …… 154
7.4.5 delegatecall 方法 …… 154
7.5 fallback 函数 …… 154
7.6 本章小结 …… 157

第8章 异常、事件与日志 …… 159

8.1 错误处理 …… 160
8.1.1 require 语句 …… 160
8.1.2 assert 语句 …… 162
8.1.3 revert 语句 …… 163

非卖品，仅供非商业用途或交流学习使用

8.2 事件与日志 …… 163
8.3 本章小结 …… 167

第9章 Truffle基础与单元测试 …… 169

9.1 应用程序开发生命周期管理 …… 169
9.2 Truffle …… 170
9.3 使用 Truffle 进行开发 …… 171
9.4 使用 Truffle 进行测试 …… 176
9.5 本章小结 …… 178

第10章 合约调试 …… 179

10.1 调试 …… 179
10.1.1 Remix 编辑器 …… 180
10.1.2 使用事件 …… 183
10.2 使用 Block Explorer …… 183
10.3 本章小结 …… 186

非卖品，仅供非商业用途或交流学习使用

第 1 章 Chapter 1

区块链、以太坊和智能合约

最近十多年来，科技和机器计算的生态系统发生了重大的变化。技术创新在多个领域影响显著，从**物联网**（IoT）到**人工智能**（AI），再到区块链（BlockChain），它们中的每一个都具有颠覆产业的力量。当前区块链已经成为最具有颠覆性的技术之一，它将潜在地改变各个行业，并将催生新的业务模式，很多行业将发生巨变。然而，区块链并不是一个全新的技术，它在过去数年间，一直在缓慢而持续地成长。区块链的突然爆发，是由于我们开始越来越多地思考去中心化和分布式应用的问题，它恰好是目前的系统架构转向不可篡改的分布式数据库的一个方案。

在第 1 章中，你将快速学习和理解一些简单的、基础性的区块链和以太坊的知识。我们也会讨论使区块链和以太坊得以运行的重要概念。同时，我们也将简要地涉及智能合约，以及如何使用 Solidity 编写智能合约。

需要注意的是，本章只会简单地介绍一些重要的区块链概念，并没有对其

非卖品，仅供非商业用途或交流学习使用

进行展开叙述，否则单单介绍概念，恐怕就需要一本书才能讲完。因为以太坊是区块链技术的实现，所以，本书中这两个词会互换使用。

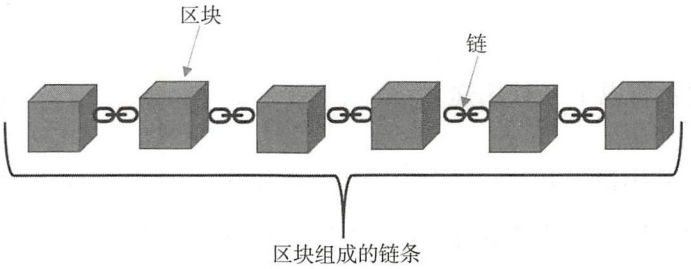
1.1 什么是区块链

区块链实质上是一个去中心化、分布式的数据库或账本，具有下列典型特征：

- **去中心化**：简单来说，在网络上一个或多个服务器瘫痪的情况下，应用或服务仍然能够持续地运行，这就是去中心化。服务和应用部署在网络上后，尽管每个服务器都有一份数据和执行程序的副本，但是没有任何一个服务器能够绝对控制数据和程序的执行过程。
- **分布式**：网络上的每个服务器或节点都互相连接在一起，服务器之间是多对多连接，而不是一对一或一对多连接。
- **数据库**：指的是存储持久化数据、用户能够及时从任何地点进行访问的地方。数据库的基本功能是数据存储和检索，同时也提供了一些管理功能，以方便高效地管理数据，如：数据导入和导出，数据备份和恢复。
- **账本**：这是一个会计专业术语。你也可以认为它是一个专门存储和检索数据的地方。账本对银行业而言很有用处。例如，Tom 在他的银行账户上存入了 100 美元，对银行而言，需要在账本上计入一笔贷方金额。未来的某一天，Tom 取回了 25 美元，银行不会直接把 100 美元修改成 75 美元，而是在同一个账本上，新增一笔借方金额 25 美元。从这个例子中可以看出，账本是一种特殊的数据存储方式，它不允许修改历史数据，要改变账户的余额只能通过新增和追加记录来实现。区块链是与账本存在共同特征的数据库，新的数据只能通过追加的方式进行存储，没有任何修改历史数据的可能。这里非常关键的一点就是理解只能通过新增记

非卖品，仅供非商业用途或交流学习使用

录去修改已有的数据。如果需要修改 100 美元的账户余额，只需要新增一笔借方或贷方记录就可以了，而不需要改变以前的记录。如下图所示：



区块链是由区块组成的一个链条。这意味着它是由多个区块前后连接在一起的，而交易记录则是保存在每个区块的内部，采用这种方式后，这些交易记录就不可能再被更改。在后面的章节中，我们会谈到交易记录的存储方式以及区块链的不可篡改特征是如何实现的。

由于去中心化和分布式特性，区块链具有稳定性、健壮性、持久性和高可用性的特点，不存在单点故障的问题。没有单个节点或服务器能控制整个链上的数据，因此人人都能够参与其中，成为区块链社区的参与者。

因为不能修改历史记录，所以区块链具有较高的可信任性、透明性和公正性。

以太坊允许通过智能合约进一步扩展它的功能，本书中将对智能合约进行详细介绍。

1.2　为什么是区块链

以太坊的主要工作原理就是接受账户发出的交易，更新它们的状态，然后维持该状态，直到另外的交易更新它们为止。在以太坊上，整个交易接受、执

非卖品，仅供非商业用途或交流学习使用

行和写入的流程分成两个阶段。以太坊接受交易和交易执行并写入账本之间是解耦的。对于去中心化和分布式架构来说，这种解耦才能够有效地支撑系统正常运行。

区块链主要有以下三种用途。

- **信任**：区块链可以用于创建去中心化应用，实现数据由多人集体控制，其中的任何一个人都没有权力去更改或删除以前的记录。即使有人确实做到了，他产生的数据也不会被其他参与者接受。
- **自治性**：对于区块链上的应用来说，没有唯一的所有者。由于没有唯一的所有者，也就没有人能够单独控制它，但是每个人却都可以通过它的行为来参与治理过程，这就有利于建立一个不能被操控或不易诱发腐败的解决方案。
- **去中介化**：基于区块链的应用能够消除现有流程的中间环节。例如在车辆登记、驾照发放等场景中，一般会存在一个中间角色，它承担着车辆登记和驾照发放的职责。如果基于区块链来设计流程，那么这个中间角色就没有存在的必要了，因为区块链上的数据在被确认后，驾照就会自动签发，车辆就会被自动登记。区块链将开启一个新的时代，很多业务不再需要中间的权威机构进行背书了。

区块链技术非常依赖于加密技术，下面我们来介绍这些内容。

1.3 加密技术

密码学是一门科学，它将易懂的、直白的内容转换成秘密的、隐藏的、无意义的内容，同样道理，解密就是反过来操作。加密技术有助于传输和保存数据，而且不能轻易地解密。

非卖品，仅供非商业用途或交流学习使用

在计算机领域有两种类型的加密技术：对称加密和非对称加密。

- **对称加密和解密**：对称加密指的是在加密和解密过程中使用同一个密钥。如果使用这种技术交换信息，那么，这意味着很多人将使用同一个密钥。
- **非对称加密和解密**：在加密和解密过程中使用两个密钥。任一密钥都可以用于加密和解密。使用公钥加密的信息，可以使用私钥解密。使用私钥加密的信息，可以使用公钥解密。下面用一个例子来说明，Tom 使用 Alice 的公钥去加密一个信息，然后发送给 Alice，然后 Alice 可以使用她的私钥解密并从中提取原始信息。简单地说，使用 Alice 的公钥加密的信息只能被她自己独有的私钥解密，这就是非对称加密的常见应用场景。非对称加密还有一个用途就是后面会介绍的数字签名。

1.3.1　散列

散列是将输入的数据转换成一个固定长度的随机字符串（散列值）的过程，但是不能从结果反向生成或识别出原始数据，因此，散列也被称为数据指纹。几乎不可能基于其散列值导出输入数据哪怕原始数据发生了一点点的变化，也将产生完全不同的散列值，这样就确保了没有人敢在原始数据上做手脚。散列还有另外一个特征：虽然输入的字符串数据可能长短不同，但产生的散列值长度是固定的。例如，使用 SHA256 散列算法，不论输入数据的长度大小如何，总会产生一个 256 个字节的散列值。当数据量很大时，这一点就非常有用了，它总能产生一个 256 个字节的散列值，这样可以保存下来作为证据。以太坊在很多地方使用了散列技术，它会对每一笔交易进行散列，会对两个交易的散列值进行再次散列，最终为同一区块内的每个交易产生一个根散列值。

散列还有一个重要特征，就是从数学上来看，两个不同的输入数据不会产生同一个散列值。同样的道理，从计算上和数学上来说都不可能由散列值反推

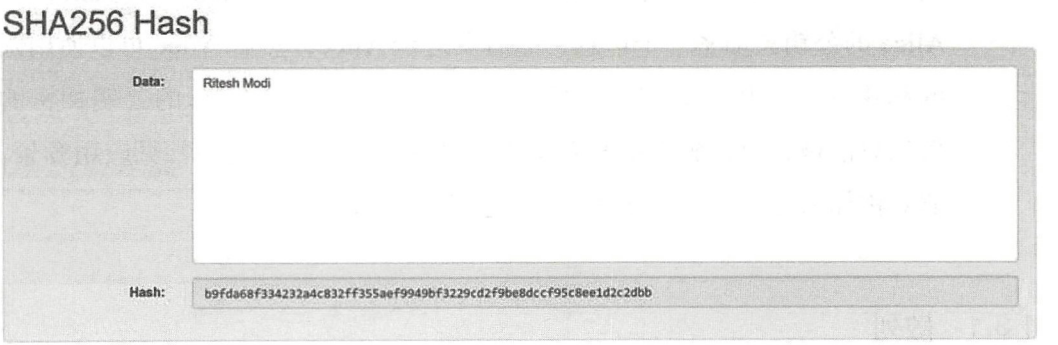
非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

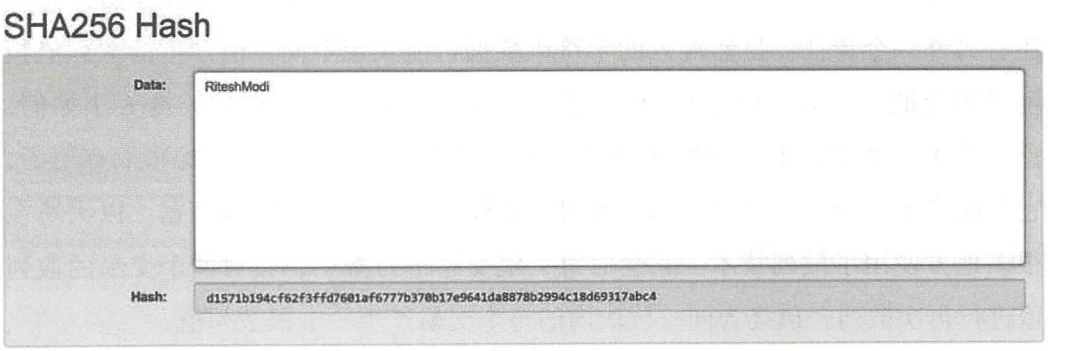
出输入值。

以太坊使用 Keccak256 作为它的散列算法。

下面的截图显示了散列的一个例子。输入的 Ritesh Modi 产生了一个散列值，如下图所示：



即使对输入数据进行轻微的变化[⊖]，也将产生一个完全不同的散列值，如下图所示：



⊖ 这里两个单词之间去掉了空格。——译者注

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

1.3.2　数字签名

前面我们介绍了非对称加密，它的一个重要应用就是在数字签名创建和验证时使用非对称密钥。数字签名类似于一个人在纸上手写的签名。与手写签名的作用一样，数字签名有助于识别一个人，还有助于确保信息在传递过程中不被篡改。让我们举个例子来理解数字签名。

Alice 准备给 Tom 发送一条信息。那么问题来了，Tom 如何确保收到的信息是由 Alice 发出来的，如何确保信息在传递过程中没有被篡改过？解决方案就是不能发送原始的信息 / 交易，Alice 首先需要取得发送的信息的散列值，然后用她的私钥对散列值进行加密，最后，她把这个刚产生的数字签名附加在散列值后发送给 Tom。Tom 收到信息后，他使用 Alice 的公钥提取出数字签名并解密，找到原始散列值。同时，他从实际接收到的信息中提取散列值，并对两个散列值进行比较，如果两个散列值一致，那么说明信息在传递过程中没有被篡改过。

数字签名通常用于资产或加密数字货币（例如以太币）的所有者对交易进行签名确认。

1.4　以太币

以太币是以太坊上的货币。以太坊的每次活动都需要消耗以太币作为费用，成功产生区块的矿工也会获得以太币作为奖励，以太币通过交易平台很容易兑换成法定货币。

以太币采用十进制的计量体系，其最小的单位是 wei。下面列出了一些计量单位，可以在网站 https://github.com/ethereum/web3.js/ blob/ 0.15.0/lib/utils/utils. js#L40 上查到更多信息。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
var unitMap = {
    'wei' : '1'
    'kwei': '1000',
    'ada': '1000',
    'femtoether': '1000',
    'mwei': '1000000',
    'babbage': '1000000',
    'picoether': '1000000',
    'gwei': '1000000000',
    'shannon': '1000000000',
    'nanoether': '1000000000',
    'nano': '1000000000',
    'szabo': '1000000000000',
    'microether': '1000000000000',
    'micro': '1000000000000',
    'finney': '1000000000000000',
    'milliether': '1000000000000000',
    'milli': '1000000000000000',
    'ether': '1000000000000000000',
    'kether': '1000000000000000000000',
    'grand': '1000000000000000000000',
    'einstein': '1000000000000000000000',
    'mether': '1000000000000000000000000',
    'gether': '1000000000000000000000000000',
    'tether': '1000000000000000000000000000000'
};
```

1.5 gas

前面介绍过，在以太坊上执行任务需要消耗以太币。由于以太币在交易所进行公开买卖，因此它的价格会上下波动。在支付费用时，如果直接使用以太币，那么，由于其价格不固定，同一个服务所花费的成本可能会忽高忽低。这样的话，人们会选择当以太币价格处于低谷时再去执行交易。对于一个平台来说，出现这种情况并不理想[⊖]。gas（燃料）的作用就是缓解这个问题，它是以太坊当前的内部货币。使用 gas 进行计价，用户能够预先确定一个交易的执行成本，这就是 **gas 成本**。采取这种方法，当以太币价格暴涨时，**gas 价格**可以适当调低，当以太币价格暴跌时，gas 价格可以适当提高。例如，使用智能合约里面的函数去修改一个字符，这个交易将会消耗一定的 gas，由于 gas 的使用成本

⊖ 会形成峰谷效应，价格低的时候，执行量过大，平台会超负荷运行。——译者注

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

已经事先确定，因此用户就可以有规律地执行智能合约了。

1.6　区块链和以太坊架构

区块链是一种包含多个组件的体系结构，区块链独特的地方在于这些组件的功能和相互作用。重要的组件包括：EVM（Ethereum Virtual Machine **以太坊虚拟机**）、矿工、区块、交易、共识算法、账户、智能合约、挖矿、以太币和gas。在本章中，我们将介绍这些组件。

一个区块链网络是由大量的节点构成的，其中一部分是属于矿工的挖矿节点，另一部分节点不挖矿但会帮助执行智能合约和交易。这些节点统称为EVM。网络上的各个节点之间互相连接，节点之间通过 P2P 协议进行通信，默认情况下使用 30303 端口。

每个节点都维护着一个账本的实例（副本），包含链上的全部区块。由于网络上存在大量矿工节点，为了避免节点之间的区块数据存在差异，这些节点会持续同步区块，确保账本数据一致。

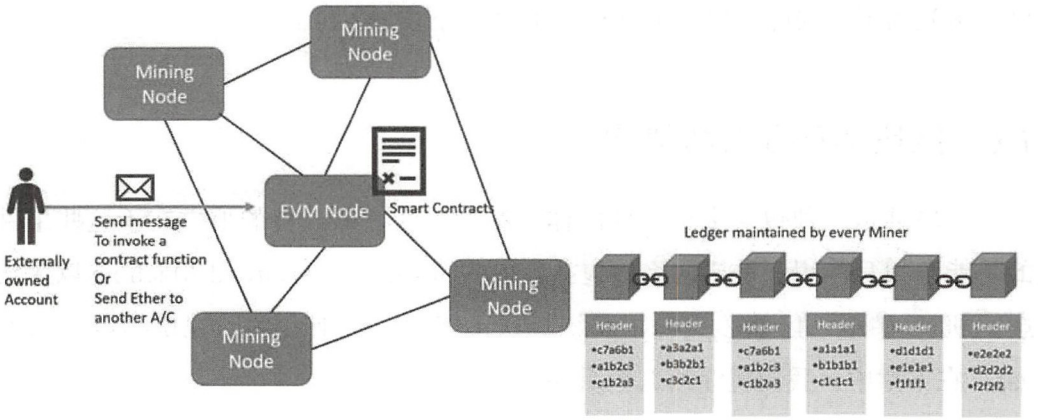
在后续的章节中，我们将会更详细地讨论账本、区块和交易。

智能合约也运行在 EVM 上。智能合约通过编写个性化的业务功能扩展以太坊的能力。智能合约执行的时候，是作为交易的一部分按照前面提到的挖矿流程进行的。

在网络上有账户的用户可以发送信息来完成账户之间的以太币交易，或者发送消息来调用合约中的一个函数。对以太坊来说，这两种方式实质上都是交易。在交易确认和更改账户余额时，账户所有人必须用私钥对交易进行数字签名，这样才能确定发送者的身份。以太坊的结构如下图所示。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！



1.6.1 区块如何互相连接

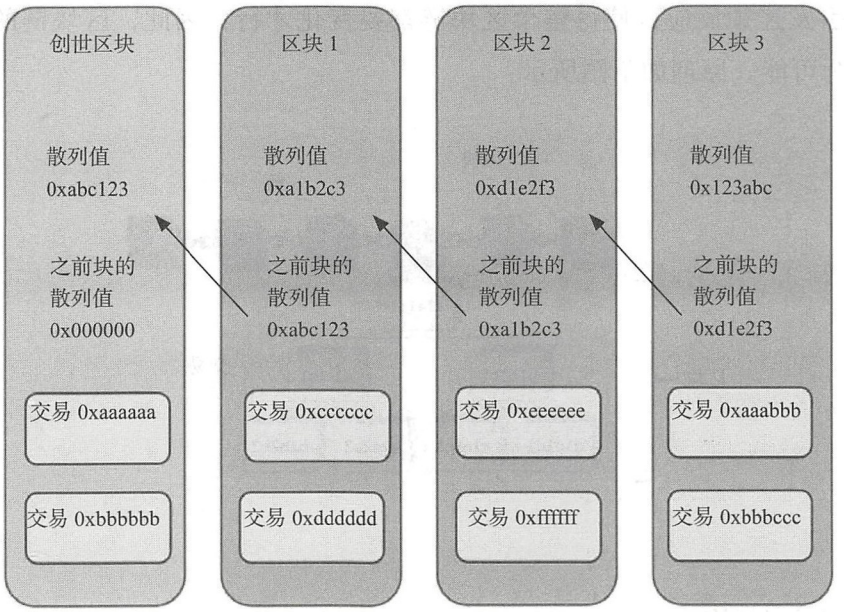
在区块链和以太坊中，每个区块都连接着另外一个区块。两个区块之间是一对父子的关系，并且是一对一的关系，这样首尾相接就组成了一个链条。本章后面会讲到区块，在接下来这张图中，我们用 3 个区块（**区块 1**、**区块 2**、**区块 3**）来示意。区块 1 是区块 2 的父区块，区块 2 是区块 3 的父区块。在每个区块的头部都存储了父区块的散列值，这样就建立了父子关系。

区块 2 在头部存储了区块 1 的散列值，区块 3 在头部存储了区块 2 的散列值，那么，问题来了，第一个区块的父区块是谁？以太坊有个**创世区块**的概念，它就是第一个区块。这个区块是在链初次发起时自动创建的。你也可以这样认为，整个链条是由创世区块（通过 `genesis.json` 文件来生成）作为第一个区块而开始启动的，如下图所示。

在后面的章节中，我们会介绍在区块链上线时，如何使用 `genesis.json` 文件创建第一个区块。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！



1.6.2　交易和区块如何互相连接

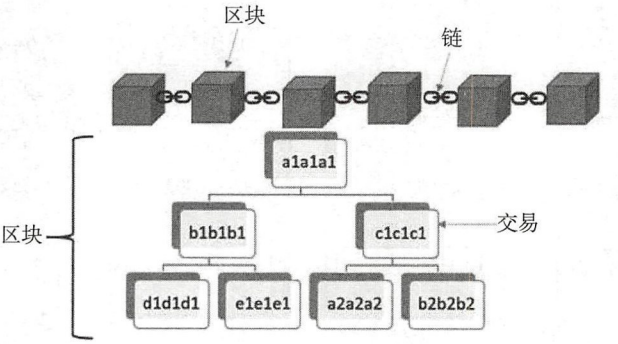
既然我们知道区块是互相连接在一起的，你可能想知道交易是如何与区块绑定在一起的。在以太坊中，交易是存储在区块里面的，交易在执行时需要消耗一定数量的 gas，由于每个区块都有一个 gas 上限，待执行的交易消耗的 gas 总数不能超过上限，这样可以避免将所有交易存储在一个区块里面。当达到 gas 上限时，其他交易就不能再写入这个区块，此时，节点开始挖矿。

一个交易在产生散列值之后，就被存储到区块中，接着，挖矿程序会选取两个交易的散列值进行再次散列，再产生一个新的散列值。很显然，区块里面的所有交易最终会产生一个唯一的散列值，它就是 **Merkle 根散列值**，保存在区块头上。如果区块里面的任何一个交易发生了改变，这个交易的散列值也会随之变化，最终会导致根交易散列值发生变化。当区块的散列值发生变化后，由于子区块保存了父区块的散列值，因此，它的子区块的散列值也需要随之变化，

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

这样会引发连锁反应，使得整个区块链都要变化才行，因此，区块链的不可篡改才成为可能。原理如下图所示：



1.7 以太坊节点

节点就是计算机，它们之间通过 p2p 协议互相连接，组成了以太坊网络。

以太坊有两种类型的节点：

- EVM
- 挖矿节点

需要注意的是，这种分类只是为了对概念进行细分，在大多数场景中，并没有专门的 EVM，相反的是，所有的矿工节点都承担了 EVM 的职能。

1.7.1 EVM

EVM 是以太坊网络的运行环境，其主要职能就是提供执行智能合约代码的环境。EVM 能够访问智能合约内部和外部的账户，以及自己存储的数据，但它只能访问当前交易内的信息，不能访问全部账本。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

EVMs 是以太坊的执行组件，其目的就是逐行地执行智能合约中的代码。但是，需要注意，当交易提交时，它实际上并没有被立即执行，而是被存放到交易池中，当然也就没有写入到以太坊账本上。

1.7.2　以太坊挖矿节点

矿工的日常工作与财务会计很相似，会计的主要工作就是在财务账本上记录数据，同样的道理，矿工的主要职责就是向以太坊写入交易数据。驱动他这样工作的原因是可以获得奖励，矿工的奖励有两种：第一种是向区块链写入区块的奖励，第二种是能够获得区块内的交易所支付的 gas 费用。一般情况下，区块链上存在很多矿工，它们之间互相竞争，然而，最终只有一个矿工能够获胜并向账本写入区块，其他矿工则无权写入。判断矿工获胜是通过破解一道难题的方式。在挖矿时，每个矿工需要解决的难题是一样的，矿工们只能通过自己的机器计算能力去破解。第一个解决了这个难题的矿工会把含有交易的区块写入自己本地的账本，然后发送区块和 Nonce 值[⊖]给其他矿工进行确认，其他矿工接收区块并验证该答案是否正确，通过后，区块就会被矿工写入到自己的账本中。在这个过程中，赢得挑战的矿工也会得到 5 个以太币的回报。每个节点都维护了一套以太坊账本的副本，矿工的职责就是通过数据同步使得本地账本处于最新的状态，最终各个矿工之间的账本实现了一致性。

矿工或者挖矿节点有三个重要的功能：

- 挖矿或打包交易产生新的区块并写入到以太坊账本
- 向其他矿工发送最新的区块，并通知他们接收
- 接收其他矿工的区块，并更新本地的账本

⊖ 破解的答案。——译者注

非卖品，仅供非商业用途或交流学习使用

挖矿节点指的是那些属于矿工的节点，它们和 EVM 节点都是整个网络的一部分。在某一时刻，矿工将产生新的区块，然后从交易池中收集交易，打包到新创建的区块中。最终，这个区块被添加到区块链上。

下面将解释一些其他概念，如共识和破解难题。

1.7.3 如何挖矿

对于网络上的全部矿工而言，挖矿程序是相同的，它们需要做的就是按照程序中的规则不断地执行挖矿任务。矿工一方面拼命计算程序以便挖到新的区块，另外，也在时刻接侦听交易池中新的交易，接收、验证和转发其他矿工的区块。正如前面所说，首先矿工创建一个区块，然后从交易池中收集交易并添加到区块中，在添加之前，会检查交易是否已经被其他矿工写入到了区块中，如果已经被写入，就去掉它。

为了获得挖矿奖励，矿工需要添加自己的币基（coinbase）交易（译者注：区块中第一笔交易是挖矿的回报交易），接着，产生区块头并执行下列步骤：

1. 矿工抓取两个交易的散列值，进行再次散列并产生一个新的散列值，直到对所有交易完成散列后，将得到唯一的散列值，这就是根交易散列值或 Merkle 根交易散列值，它将被添加到区块头。

2. 矿工也需要确定上一个区块的散列值，因为它是当前区块的父区块，父区块的散列值要保存在当前区块头中。

3. 矿工计算交易的 state 和 receipts 根散列值[⊖]，然后写入区块头。

4. Nonce 和时间戳记录到区块头。

⊖ 这里的 state 和 receipts 指的是另外两个 Merkle 树。以太坊有三棵 Merkle 树，即：交易 Merkle 树，state Merkle 树和收据 Merkle 树。——译者注

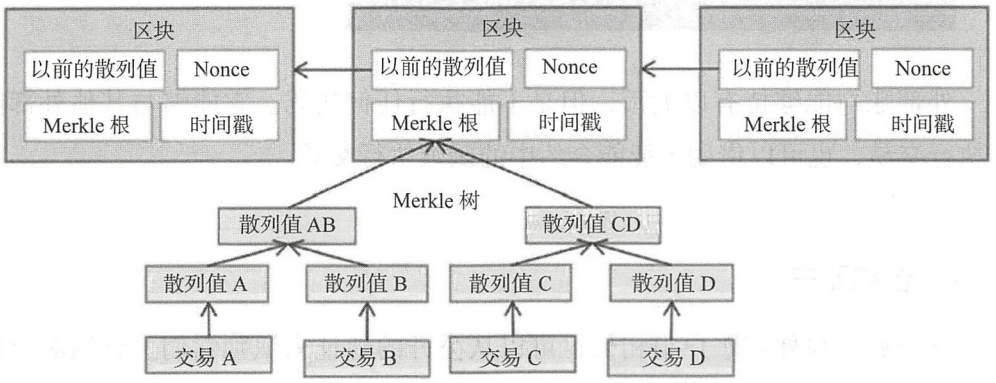
非卖品，仅供非商业用途或交流学习使用

5. 产生整个区块（包括区块头和区块体）的散列值。

6. 挖矿流程开始，矿工持续变换 Nonce 值，直到发现该散列值能够解决难题为止，需要记住的是，对网络上的矿工而言，执行过程是一样的。

7. 很显然，某一矿工最终能够找到这个难题的答案，它会将结果发送给网络上的其他矿工。其他矿工会先确认该答案是否正确，如果是正确的，将开始验证每个交易，然后接受该区块，并添加到他们的本地账本中。

上面的整个挖矿流程，因为矿工提供了经过不断计算而获得了解题答案的证明，所以，被称之为**工作量证明（PoW）**。另外，还有其他一些算法，如：**权益证明（PoS）**和**权威证明（PoA）**，由于以太坊没有用到这些算法，本书暂不展开讨论。区块头和它的组成部分如下图所示：



1.8　以太坊账户

账户是以太坊体系的主要组成部分。以太坊将交易保存到账本上的过程，其实就是账户之间进行交互的过程。以太坊有两种类型的账户：外部账户和合约账户。每个账户默认有一个余额属性，可以查看该账户以太币的当前余额。

1.8.1 外部账户

以太坊上的用户所拥有的账户。在以太坊中，账户不能使用名称来调用。当你在以太坊上创建一个外部账户时，会产生一对公钥和私钥。私钥需要你保存在安全的地方，而公钥就是你对账户持有所有权的证明。公钥一般是256个字符，但是以太坊只使用前面160个字符来标识身份。

例如，Bob在以太坊网络上（无论是公开网络还是私有网络）新建一个账户，私钥留给自己使用，而公钥的前160位字符将成为它的身份标识，其他账户就可以向这个账户发送以太币或者其他基于以太的加密货币。

以太坊上的账户类似于下面截图中的样式

```
0xa57de277ede9c1521f51f6989ed2497a5b9c1926
```

外部账户能够持有以太币，但是不能执行任何代码。它能够与其他外部账户执行交易，也可以借助于智能合约中的函数执行交易。

1.8.2 合约账户

合约账户与外部账户很相似，可以从公开的地址来识别它们。合约账户没有私钥，但可以像外部账户一样持有以太币，不同的是，合约账户可以包含代码，它的代码是由函数和状态变量组成的。

1.9 交易

交易是买方和卖方、供应商和消费者、提供方和消费方之间使用货币、加密数字货币或者其他资产换取对方的资产、产品或服务的协议。以太坊为交易

顺利执行提供了保障，下面是以太坊上支持的三种交易类型：

1. **从一个账户向另外一个账户发送以太币**：这个账户可能是外部账户或者合约账户。下面的场景都可能发生：

- 在交易中，一个外部账户向另外一个外部账户发送以太币
- 在交易中，一个外部账户向一个合约账户发送以太币
- 在交易中，一个合约账户向另外一个合约账户发送以太币
- 在交易中，一个合约账户向一个外部账户发送以太币

2. **智能合约部署**：外部账户在 EVM 上部署合约是通过交易的方式实现的。

使用或借助合约内的函数：如果需要执行合约内的函数去改变一个状态，就需要一个交易，如果执行函数没有改变任何一个状态，就不需要交易。

下面介绍与交易有关的一些重要属性：

`From` 账户属性说明了这个账户是交易的发起方，发送 gas 或以太币。前面章节我们介绍过以太币和 gas 的概念。`From` 账户可以是外部账户或合约账户。

`To` 账户属性指的是接收以太币或其他收益的账户，它可以是外部账户或合约账户。如果是部署合约的交易，则 To 字段为空。

`Value` 账户属性指的是账户之间转移的以太币数量。

`Input` 账户属性指的是合约编译后被部署在 EVM 上的字节码。`input` 也用于保存有关智能合约函数带参调用的信息。下图展示了在典型的以太坊交易中使用智能合约函数的地方，从这个截图上看，请注意 `Input` 字段中包含了带有参数的函数调用。

```
{ blockHash: '0xba93a91df520c7565e80347346e47b83a41d473a33352d1cf7e689c30b305ba5',
  blockNumber: 70,
  from: '0xa57de277ede9c1521f51f6989ed2497a5b9c1926',
  gas: 90000,
  gasPrice: BigNumber { s: 1, e: 10, c: [ 18000000000 ] },
  hash: '0x6b65b86462e6aa89d5f9469ce03b9ea21e8bf72f8c11aa72de2978f9b7a5b9fd',
  input: '0xc8aaea400000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000
  nonce: 1,
  to: '0x6b90c690b23af11c9575c2b9b8e26d47d84b4f8b',
  transactionIndex: 0,
  value: BigNumber { s: 1, e: 0, c: [ 0 ] },
  v: '0x42',
  r: '0xd4f5adb0f1739105668afa5aba700ae4f031e88e9f21bdb7ac787a3af156baf7',
  s: '0x7f89cda2ae325948f73bcd8da93e2cd6cf9917b22e32d93c3f8caedc0edcd1' }
```

BlockHash 账户属性指的是该交易所属的区块的散列值。

BlockNumber 账户属性指的是交易所属的区块序号。

Gas 账户属性指的是交易的发送方支付的 gas 数量。

GasPrice 账户属性指的是发送方支付的 gas 价格，以 wei 为单位（在本章前面介绍以太币的地方，提到过 wei 的概念）。总的 gas 消耗 =gas 数量 *gas 价格。

Hash 账户属性指的是交易的散列值。

Nonce 账户属性指的是交易的编号，它由发送方在当前交易之前产生。

TransactionIndex 账户属性指的是区块中当前交易的流水号。

Value 账户属性指的用 wei 计算的传递的以太的数量

V、r 和 s 属性指的是数字签名和交易的签名。

```
{ blockHash: '0x78ddc6d1d18a52811888dea659a69f35f424aa0ec48562b956d3524e80fcf893',
  blockNumber: 105,
  from: '0xa57de277ede9c1521f51f6989ed2497a5b9c1926',
  gas: 90000,
```

```
gasPrice: BigNumber { s: 1, e: 10, c: [ 18000000000 ] },
hash: '0x93768f05999d54edde1982f82150b429b3cba0014233defab34701e6b6a7ec87',
input: '0x',
nonce: 2,
to: '0x9d2a327b320da739ed6b0da33c3809946cc8cf6a',
transactionIndex: 0,
value: BigNumber { s: 1, e: 18, c: [ 20000 ] },
v: '0x41',
r: '0x9efb14382840ab5fcdf2d33f32638e895beb9cee35d4d79675c183c7fddef8f5',
s: '0x658bac95226e3a8a90d497ce8c841e0833c01b5a2567bb8c2aa126ba95e1fbd2' }
```

下面是以太坊上的一个常见的交易，外部账户向另外一个外部账户发送以太币（注意这里没有使用 input 字段）。截图的交易中转移了 2 个以太币，其中的 value 字段就是用 wei 来计量的。

外部账户之间转移以太币也可以使用下面的代码来实现，它基于的是 web3 JavaScript 框架，本书后面会介绍。

```
web.eth.sendTransaction({from: web.eth.accounts[0], to:
"0x9d2a327b320da739ed6b0da33c3809946cc8cf6a", value: web.toWei(2,
'ether')})
```

下图展示了部署了智能合约的交易，注意 input 字段中包含了合约编译后的字节码。

```
{ blockHash: '0x041cdd69390b130e0b54c53f2afb46d79a06708dcf8414aa1ba4bbb40a2786b7',
  blockNumber: 6,
  from: '0xa57de277ede9c1521f51f6989ed2497a5b9c1926',
  gas: 1000000,
  gasPrice: BigNumber { s: 1, e: 10, c: [ 18000000000 ] },
  hash: '0x6f5a74e5191f745d0e38fa67841ba6b36cf03a7c0fd7729ef355fe77c86487a2',
  input: '0x60606040523415610000f57600080fd5b6102c38061001e6000396000f300606060405260043610610041b5763ffffffff7c0100
561005b57600080fd5b61006361012d565b604051602080825281908101838181518152602001915080519060200190808383600005b8381101
0191505b509250505060405180910390f35b34156100e557600080fd5b61012b600460248135818101908301358060206001f82018190048102
6156101000203166002900480601f0160208091040260200160405190810160405280929190818152602001828054600181600116156101000
0190602001808311610 1ae5782900360 1f168201915b505050505090505b90565b60008180516101e992916020019061 01ff565b5050565b60
1024057805160ff191683800117855561026d565b828001600101855582156102 6d579182015b828111156102 6d57825825591602001919067
7752e38ed74bb682568cb64c68f24734ab95e18740deee526b95eff79e40029',
  nonce: 0,
  to: null,
  transactionIndex: 0,
  value: BigNumber { s: 1, e: 0, c: [ 0 ] },
  v: '0x42',
  r: '0x29887013743c6fc9a4bb78c6d3ca2974eac6f41b21d2d0bb6fdfe09b71202602',
  s: '0xaaa9366553495c9c81ecb9806d0cdcb4e0ed5d688797be099978e260ae53e34' }
```

1.10 区块

区块是以太坊的一个重要概念。区块是交易的容器，它由多个交易组成。因为受区块的 gas 上限（后面我们会介绍 gas 上限）和区块大小限制，每个区块包含的交易数量并不相同。区块连接在一起形成了区块链，除了第一个区块（也叫作创世区块）没有父区块外，其他区块都有一个父区块，父区块的散列值存储在区块头上。

下图显示了区块的结构：

```
{ difficulty: BigNumber { s: 1, e: 5, c: [ 135070 ] },
  extraData: '0xd783010702846765746885676f312e398777696e646f7773',
  gasLimit: 4011042861,
  gasUsed: 43406,
  hash: '0xba93a91df520c7565e80347346e47b83a41d473a33352d1cf7e689c30b305ba5',
  logsBloom: '0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000',
  miner: '0xa57de277ede9c1521f51f6989ed2497a5b9c1926',
  mixHash: '0x4e80de770c329aebbc2e9e861190784f57b7f9910bbb049534828052284f32e4',
  nonce: '0x655dee191333922c',
  number: 70,
  parentHash: '0x27d3dbc34614f88583f29ea1b7546e83563e55638b1d0a258de04f4912f42aaf',
  receiptsRoot: '0x5dff465dd85c4ad02c71ec4099284ecaf91ed9bb600f7903978386059c16fb8d',
  sha3Uncles: '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
  size: 742,
  stateRoot: '0xb15363a8958d218eff295b5e877517ee4243f1b681d987793c5ec30a04bc4592',
  timestamp: 1511421241,
  totalDifficulty: BigNumber { s: 1, e: 6, c: [ 9302609 ] },
  transactions:
   [ '0x6b65b86462e6aa89d5f9469ce03b9ea21e8bf72f8c11aa72de2978f9b7a5b9fd' ],
  transactionsRoot: '0x5aceca068d1a7ac8d8ecd8f19469ec7c687cb6ceed54e0db39b58dfdf6481de9',
  uncles: [] }
```

区块有很多属性，为了便于掌握关键内容，下面只介绍一些重要的部分：

`Difficulty` 属性指的是矿工为了挖到这个区块而需要面对的计算难度。

`GasLimit` 属性指的是区块允许的 gas 总量上限。它决定了区块中能包含多少个交易。

`GasUsed` 属性指的是区块中的交易实际消耗的 gas 数量。

`Hash` 属性指的是这个区块的散列值。

`Nonce` 属性指的是一个数字，它是解决难题的答案。

`Miner` 属性指的是矿工的账户，可以用 coinbase 或 etherbase 的地址。

`Number` 属性指的是该区块在区块链上的序号。

`ParentHash` 属性指的是父区块的散列值。

`ReceiptsRoot`、`stateRoot` 和 `TransactionsRoot` 属性指的是在前面的挖矿流程中提到的 merkle 树

`Transactions` 属性指的是区块中的交易组成的一个数组。

`TotalDifficulty` 属性指的是区块链的整体难度。

1.11　端到端的交易

前面介绍了区块链和以太坊的一些基本知识，接下来我们介绍一个完整的端到端的交易流程，看看交易如何贯穿多个组件并保存到账本中。

本例中，Sam 打算发送一个数字资产（如：美元）给 Mark。首先，Sam 新建了一个交易，里面包括了 from、to、value 等字段数据，然后发送到以太网络上，该交易并没有立即写入到账本中，而是暂存到交易池中。

挖矿节点新建了一个区块，然后按照 gas 上限标准，从交易池中提取交易（Sam 的交易也将被提取），并添加到区块中，网络上的全体矿工都在执行相同

的任务。

接下来，矿工们开始争先恐后地去计算难题，在一段时间（或几秒）后，获胜者（第一个解决难题的人）会发出通知，声称他找到了答案，赢得了比赛，需要向区块链写入区块，与此同时，获胜者将答案添加到区块上并发送给其他矿工。其他矿工收到通知后，首先验证这个答案，一旦认定该答案确实有效，就立即停止自己的计算，接收这个包含了 Sam 的交易的区块，然后添加到他们的本地账本中。这样下来，就在链上产生了一个新的区块，它将一直跨越时间和空间而永久的存在下去。在这期间，双方的账户余额都会得到更新，最后，区块被分发复制到网络上的全部节点。这个过程如下图所示：

1.12 什么是合约

合约是经过双方或多方同意，约定立即执行或在将来执行一项交易的法律文件。因为合约是法律文件，所以它具有强制性和可执行性。合约应用的场景很多，例如：一个人和保险公司签订合同购买健康险，一个人从另外一个人手里购买一块土地，一个公司出售股权给另外一家公司。

1.13 什么是智能合约

智能合约是按照用户的需求编写的代码，并部署和运行在以太坊虚拟机

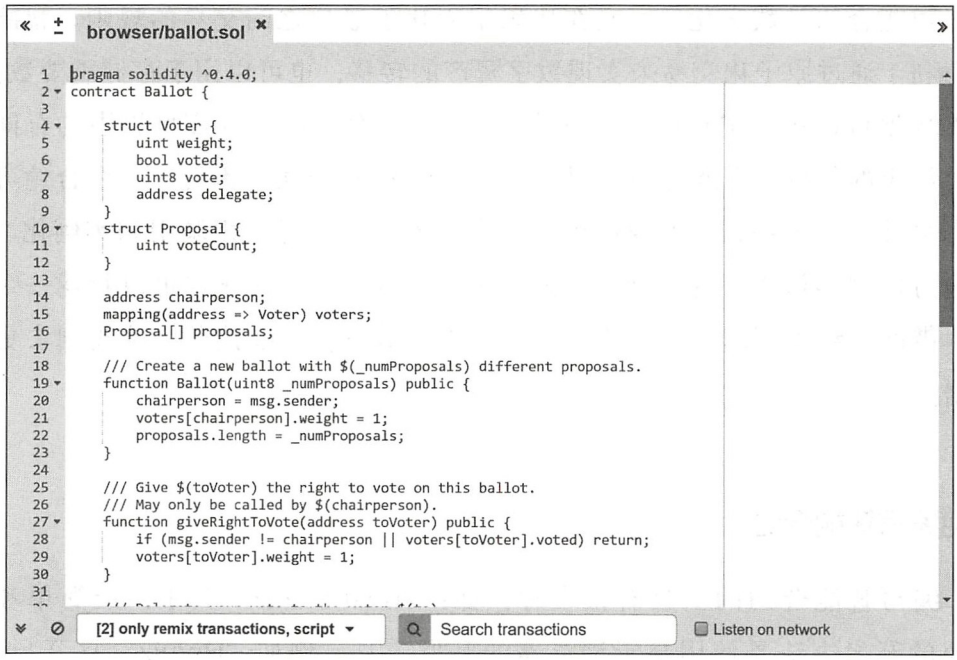
上。智能合约是数字化的，它在代码中固化了账户之间交易的规则。智能合约有利于通过原子化交易来实现数字资产的转移，也可以用于存储重要数据，这些数据可以用来记录信息、事件、关系、余额、以及现实世界中的合同中需要约定的信息。智能合约类似于面向对象的 class 类，因此，一个合约可以调用另外一个合约，就像我们可以在类对象之间进行互相调用和实例化。我们也可以这样认为，智能合约就是由函数构成的小程序。你可以新建一个合约，借助合约里面的函数去查看区块链上的数据，以及按照一些规则去更新数据。

如何编写智能合约

编写智能合约的工具有很多种，如：Visual Studio。其中，最简单和最快速的开发方法是使用基于浏览器的开发工具，例如：Remix。打开 http://remix.ethereum.org 网页就可以直接使用。Remix 是一个新名字，以前被叫做 browser-solidity。Remix 提供了丰富的 Solidity 集成开发环境，可以在浏览器上进行智能合约的创建、开发、部署和调试。合约维护有关的操作（如：创建、发布、调试）都可以在同一个环境下完成，而不需要切换到其他的窗口或页面。

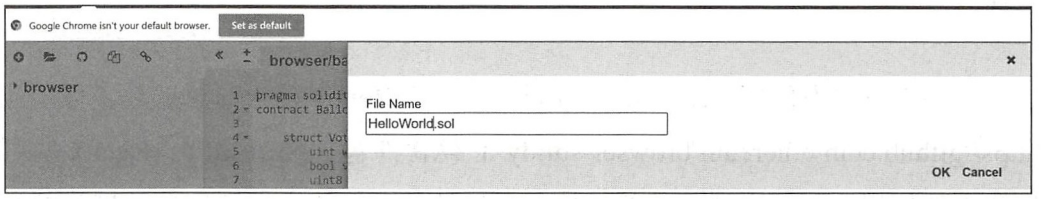
有些人不习惯使用 Remix 的在线版本。由于 Remix 是个开源工具，可以从 https//github.com/ethereum/browser-solidy 下载软件包，经过编译，Remix 就可以在本地使用。使用本地工具的另外一个好处是它能够连接到你自己搭建的私有网络，否则的话，你在网页上编写完代码后，还需要把文件拷贝到本地，经过编译后才能发布到私有网络上。下面我们来逐步了解 Remix。

1. 打开 remix.ethereum.org 网址，在浏览器中默认打开一个智能合约（见下图），如果你不需要，可以删掉它。



2. 新建一个合约，选择左边菜单栏中的 +。

3. 对这个 Soldiy 文件进行命名，以 .sol 作为后缀。输入合约名字 HelloWorld，点击“OK”，就创建了一个空白合约，如下图所示：



4. 在制作栏内的空白处，输入下面这段代码，就能创建你的第一个合约。

合约的细节会在第三章讲解。现在，先简单理解一下，你可以使用关键词 contract 创建合约，声明全局状态变量和函数，保存合约为后缀名 . 是 sol 的文件。在下面的源代码片段中，当 GetHelloWorld 函数调用 HelloWorld 合

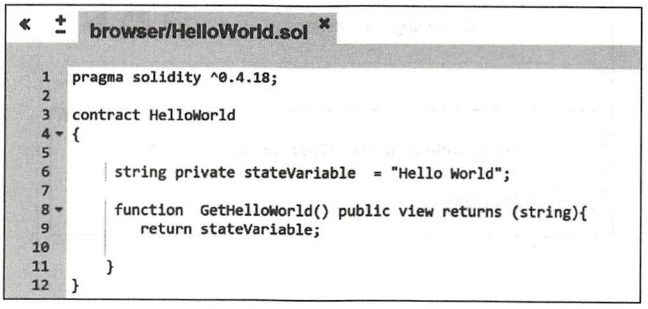
严禁网络传播本PDF，违者责任自负！

约时，将返回 Hello World 字符。

```
pragma Solidity ^0.4.18;
contract HelloWorld
{
string private stateVariable = "Hello World";
function GetHelloWorld() public view returns (string)
{
return stateVariable;
}
}
```

Remix 的右面是操作窗口，有很多标签页面：**编译**、**运行**、**设置**、**调试**、**分析**和**支持**。这些操作用于对合约进行编译、发布、调试和调用。调试标签会把合约编译成字节码（以太坊能够理解的代码），这里会显示调试时出现的警告和错误信息，对于这些信息，你要足够重视，解决这些信息会增强合约的健壮性。运行页面与编写合约页面相比，你需要花费更多的时间。在“环境”选项中，由于 Remix 在浏览器中绑定了以太坊的运行环境，所以允许你使用 JavaScript 虚拟机对合约进行发布，Injected Web3 选项用于使用 Mist 和 MetaMask 工具（下一章会提到），Web3 Provider 选项是在本地使用 Remix 连接私有链时使用的。在本章的例子中，默认使用 JavaScript 虚拟机选项。在第三章介绍 Solidity 时，再讨论剩下的选项。

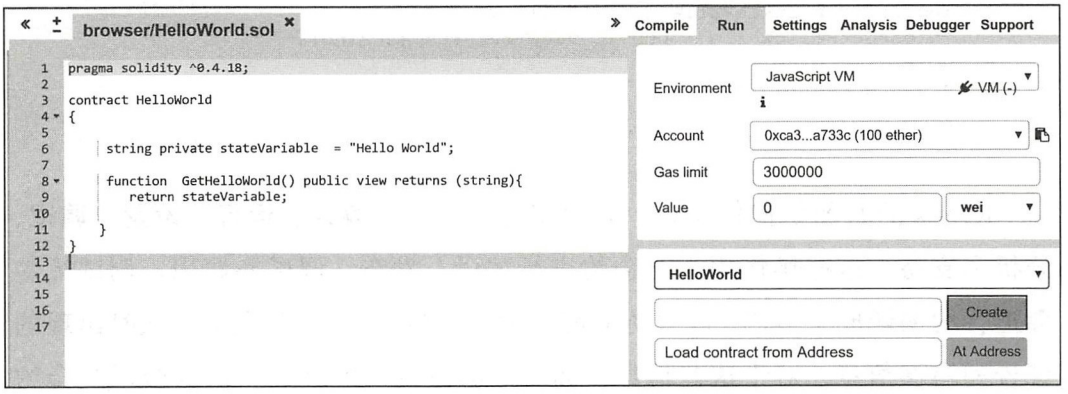
5. 然而，发布合约最重要的操作就是使用**新建**按钮，如下图所示：



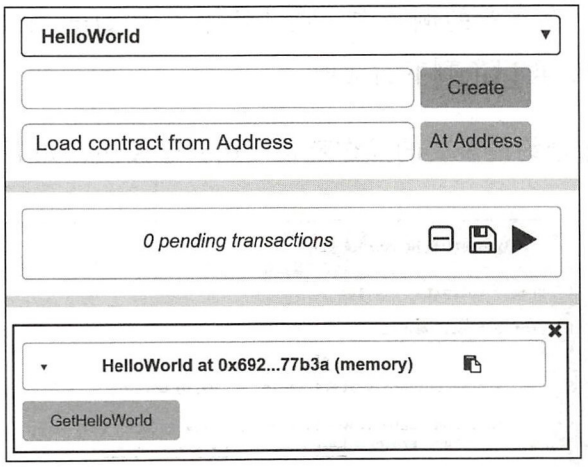
6. 点击**新建**按钮，就可以在浏览器中运行以太坊，在**新建**按钮下面显示了

严禁网络传播本PDF，违者责任自负！

合约内的函数。由于例子中只有唯一的函数 GetHelloWorld，所以这里只有这一个选项。如下图所示：



7. 点击 GetHelloWorld 按钮，就可以调用和执行这个函数。执行的结果显示在 Remix 下面的栏位，如下图所示：



恭喜！你的第一个合约已经创建、发布和执行成功了。如果你不喜欢直接敲代码的话，本章中的 `HelloWorld` 合约代码可以拷贝到 Remix 中直接使用。

严禁网络传播本PDF，违者责任自负！

1.14　如何部署合约

Remix 让发布过程变得简单了，但是系统后台还是执行了很多程序。了解这些步骤有助于我们更好地掌控发布过程。

第一步是使用 Solidity 编译器进行合约编译（下一章我们将介绍如何下载和使用 Solidity 编译器）。

编译后主要有两种输出物：

- ABI 规范
- 合约字节码

ABI（Application Binary Interface）是一个接口，由带有参数的外部函数和公共函数组成。合约和其他使用者如果准备调用合约里面的函数，就可以使用 ABI 来实现。

字节码是合约的体现形式，它运行在以太坊上面。在发布时，字节码是必须的，ABI 只有在调用合约里面的函数时才会用到。你可以使用 ABI 创建一个新的合约实例。

合约的发布本身就是一个交易。因此，为了发布合约，你需要新建一个交易。在发布时，需要提供字节码和 ABI。由于交易在运行时需要消耗 gas，这些 gas 就需要由合约来提供。一旦交易被打包写入到区块链上后，你就可以通过合约地址来使用合约了，调用方也可以通过新地址来调用合约里面的函数。

1.15　本章小结

本章介绍了区块链，重点介绍了以太坊。对区块链和以太坊的工作原理深

严禁网络传播本PDF，违者责任自负！

入理解，会有助于你使用 Solidity 编写更为健壮、安全和高效的智能合约。本章介绍了区块链的基本概念，解释了区块链是什么，区块链为什么这么重要，对于构建去中心化和分布式应用有哪些价值。本章也简要地介绍了区块链的体系结构，以及一些重要的概念，如：交易、区块、gas、以太币、账户、密码学和挖矿。本章也初步涉及智能合约的一些内容，例如使用 Remix 去编写智能合约，以及如何运行它。本章介绍的比较简单，因为后面会进一步详细的阐述，那时候你就能利用 Solidity 开发智能合约了。

你会注意到本章并没有提及以太坊的一些工具。下一章将介绍安装以太坊和相关工具，以太坊的生态十分丰富，有很多工具可以利用。我们将选取一些重点进行介绍，如：`web3.js`、TestRPC、Geth、Mist 和 MetalMask。

严禁网络传播本PDF，违者责任自负！

第 2 章 Chapter 2

安装以太坊和 Solidity

在第 1 章中，我们介绍了所有有关区块链的概念，主要关注了以太坊，也解释了区块链运行的原理。在部署以太坊的解决方案时，可以选择不同类型的网络，如：公共网络、测试网络和私有网络。本章将重点介绍搭建以太坊环境所需要的工具。在以太坊生态中，工具有很多种，我们将介绍其中一些比较重要的。本书中我们的部署环境是 Azure 云的 Windows server2016。当然，它们也能部署在 Linux、Mac 和其他虚拟机或物理机上。在环境搭建好以后，我们就可以使用 Solidity 进行智能合约的编写、测试和发布了。

2.1 以太坊网络

以太坊网络是一个创建和部署分布式应用的开源平台，依靠大量计算机（节点）支撑，它们之间相互交互，并在分布式账本中存储数据。在这里，分布式账本的含义是网络上的每个节点都拥有一个账本的副本。对开发者来说，有

严禁网络传播本PDF，违者责任自负！

多种网络部署方式可供选择，在部署解决方案和智能合约时，可以根据实际需求和场景，选择一个适合自己的网络，这样就不会支付真实的以太币或发生其他支出。当然，也有免费的网络可以使用，也有网络需要用户支付以太币或其他货币。

2.1.1 主网

以太坊主网是全球性的公开网络，人人都可以使用，通过账户就能进入网络。对于任何人来说，创建账户、部署解决方案和合约都是免费的。主网的使用费用以 gas 来计量。它的代号是 homestead（以前叫做 Frontier）。通过互联网，人们能连接上主网，并可以查看上面的数据和交易。

2.1.2 测试网络

测试网络的目的是帮助人们快速适应和使用以太坊环境，是从主网精确复制而来。在测试网络上，部署和使用合约，都不会发生真实的费用，这是因为测试用的以太币都是任意产生的，只能在测试网上使用。在笔者编写此书时，存在多个测试网络可以使用，例如：Ropsten、Kovan 和 Rinkeby。

Ropsten 是第一个使用 PoW 共识算法产生区块的测试网络。以前被叫做：Morden。正如上面所说的，你可以无偿使用它构建和测试合约。在 Geth 中加入——`testnet` 参数就可以进入测试网络了。在后面会介绍 Get 的用法。Ropsten 是目前为止最受欢迎的测试网络。

Rinkeby 是另外一个以太坊测试网络，它使用的是 PoA 共识机制。PoW 和 PoA 在矿工间建立共识时的机制是不同的。PoW 在维护不可篡改性和去中心化方面更强壮，然而它的缺陷是不能有效地控制矿工。PoA 具备了 PoW 的优点，

严禁网络传播本PDF，违者责任自负！

也能对矿工具有一定的控制能力。

Kovan 测试网络只能在少部分用户之间使用，因此，本书没有用到它，也不会介绍。更多的信息可以访问：https://kovan-testnet.github.io/website/。

2.1.3　私有网络

私有网络是在用户自有的网络上建立和运行的，控制权掌握在一个组织手里。出于测试的考虑，人们并不希望将解决方案、合约和场景放到公共网络上，这样就需要建立一个开发、测试和生产的环境，因此，用户应该搭建一个私有网络，这样就能够全盘进行掌控。在下面的章节中，我们将看到如何搭建。

2.1.4　联盟网络

联盟网络也是一个私有网络，只是有一点区别，联盟网络的节点是由不同组织所管理的。实际上，联盟链上也没有哪个组织能够单独地控制网络和数据，而是由全体组织和组织里面具有查看和修改权限的人共同进行控制的。联盟链可以通过互联网或 VPN 网络接入。

2.2　Geth

目前存在多种语言编写的以太坊客节点和客户端工具，包括：Go、C++、Python、javaScript、Java 和 Ruby 等。这些功能不受语言限制，开发者可以选择最适合自己的。本书使用 Go 语言，它被称为 Geth，可以作为客户端连接到公共的和测试的网络上，它也可以在私有网络上运行挖矿和 EVM（交易节点）。

严禁网络传播本PDF，违者责任自负！

Geth 是使用 Go 语言编写的命令行工具，可以使用它在私有网络上创建节点和矿工。Geth 可以在 Windows、Linux 和 Mac 环境下安装。

在 Windows 下安装 Geth

在搭建私有网络前，需要下载和安装 Geth（goethereum）工具。

下面介绍下载和安装步骤：

1. 可以访问 https://ethereum.github.io/go-ethereum/downloads/ 页面下载，有 32 位和 64 位两个版本。在本书中，操作系统使用 Azure 云上的 windows Server 2016 版本。

2. 下载后，执行安装文件，接下来按照默认配置一步步进行，完成开发环境设置。

3. 安装完成后，就可以在 CMD 和 Powershell 中使用了。

4. 打开命令行，输入 `geth –help`

需要注意的是，在输入 geth 后，get 会连接到主网上，然后开始同步和下载区块和交易数据。目前，整个以太坊区块已经超过 30GB[⊖]。help 命令执行后，屏幕上会显示 geth 可用的命令，包括版本号。

⊖ 这是作者编写本书时的情况，截止到 2018 年 6 月底，全部区块已经超过 1TB。——译者注

严禁网络传播本PDF，违者责任自负！


```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\citynextadmin>geth help
NAME:
   geth - the go-ethereum command line interface

   Copyright 2013-2017 The go-ethereum Authors

USAGE:
   geth [options] command [command options] [arguments...]

VERSION:
   1.7.2-stable-1db4ecdc

COMMANDS:
   account     Manage accounts
   attach      Start an interactive JavaScript environment (connect to node)
   bug         opens a window to report a bug on the geth repo
   console     Start an interactive JavaScript environment
   copydb      Create a local chain from a target chaindata folder
   dump        Dump a specific block from storage
   dumpconfig  Show configuration values
   export      Export blockchain into file
   import      Import a blockchain file
   init        Bootstrap and initialize a new genesis block
   js          Execute the specified JavaScript files
   license     Display license information
   makecache   Generate ethash verification cache (for testing)
   makedag     Generate ethash mining DAG (for testing)
   monitor     Monitor and visualize node metrics
   removedb    Remove blockchain and state databases
   version     Print version numbers
   wallet      Manage Ethereum presale wallets
   help, h     Shows a list of commands or help for one command
```

Geth 是基于 JSON RPC 的协议。它定义了采用 JSON 格式的代码远程调用规范。Geth 可以使用下面三种 JSON RPC 协议进行连接：

- **内部进程通信**（IPC，Inter Process Communication）：内部通讯，通常用于一台电脑内。
- **远程程序调用**（RPC，Remote Procedure Calls）：跨计算机的通讯。通常使用 TCP 和 HTTP 协议。
- **WS，Web sockets**：使用 sockets 连接 Geth。
- 配置 Geth 有很多命令、开关参数和选项，包括：

严禁网络传播本PDF，违者责任自负！

- 配置 IPC、RPC 和 WS 协议
- 配置网络连接的类型——私有网络、Rostper 和 Rinkeby
- 挖矿选项
- 控制台和 API
- 网络
- 调试和日志

其中，在创建私有网络时用到的一些选项，将在下一部分提到。

Geth 可以直接使用 `Geth`（不带选项）直接连接公链。Homestead 就是当前公链的名称。它的 `Networkid` 和 `ChainID` 是 1。

```
C:\Users\citynextadmin>Geth
WARN [11-13|07:53:00] No etherbase set and no accounts found as default
INFO [11-13|07:53:01] Starting peer-to-peer node               instance=Geth/v1.7.2-stable-1db4ecdc/windows-amd64/go1.9
INFO [11-13|07:53:01] Allocated cache and file handles         database=C:\\Users\\citynextadmin\\AppData\\Roaming\\Ethe
reum\\geth\\chaindata cache=128 handles=1024
INFO [11-13|07:53:01] Initialised chain configuration          config="{ChainID: 1 Homestead: 1150000 DAO: 1920000 DAOSu
pport: true EIP150: 2463000 EIP155: 2675000 EIP158: 2675000 Byzantium: 4370000 Engine: ethash}"
INFO [11-13|07:53:01] Disk storage enabled for ethash caches   dir=C:\\Users\\citynextadmin\\AppData\\Roaming\\Ethereum\
\geth\\ethash count=3
INFO [11-13|07:53:01] Disk storage enabled for ethash DAGs     dir=C:\\Users\\citynextadmin\\AppData\\Ethash
          count=2
INFO [11-13|07:53:02] Initialising Ethereum protocol           versions="[63 62]" network=1
INFO [11-13|07:53:02] Loaded most recent local header          number=0 hash=d4e567…cb8fa3 td=17179869184
INFO [11-13|07:53:02] Loaded most recent local full block      number=0 hash=d4e567…cb8fa3 td=17179869184
INFO [11-13|07:53:02] Loaded most recent local fast block      number=0 hash=d4e567…cb8fa3 td=17179869184
INFO [11-13|07:53:02] Loaded local transaction journal         transactions=0 dropped=0
INFO [11-13|07:53:02] Regenerated local transaction journal    transactions=0 accounts=0
INFO [11-13|07:53:02] Starting P2P networking
```

在不同的网络上，其 Chain ID 是不同的。其中：

- chain ID 1 是主网公链
- chain ID 2 是 Morden 网络（仅对部分人开放）
- chain ID 3 是 Ropsten 网络
- chain ID 4 是 Rinkeby 网络
- chain ID 大于 4 的是私有网络。

另外，可以使用 Geth--testnet 连接到 Ropsten 网，——rinkeby 连接到 Rinkeby 网，Geth 也可以与 chain ID 配合使用。

2.3　搭建一个私有网络

Geth 安装好后，可以先在本地进行配置，这时不需要连接到互联网。每个网络都有一个创世区块，它是整个网络的第一个区块，没有父区块。创世区块由 genesis.json 文件产生。下面的代码显示了 genesis.json 的内容。

```
{
"config": {
"chainId": 15,
"homesteadBlock": 0,
"eip155Block": 0,
"eip158Block": 0
},
"nonce": "0x0000000000000042",
"mixhash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
"difficulty": "0x200",
"alloc": {},
"coinbase": "0x0000000000000000000000000000000000000000",
"timestamp": "0x00",
"parentHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
"gasLimit": "0xffffffff",
"alloc": {
}
}
```

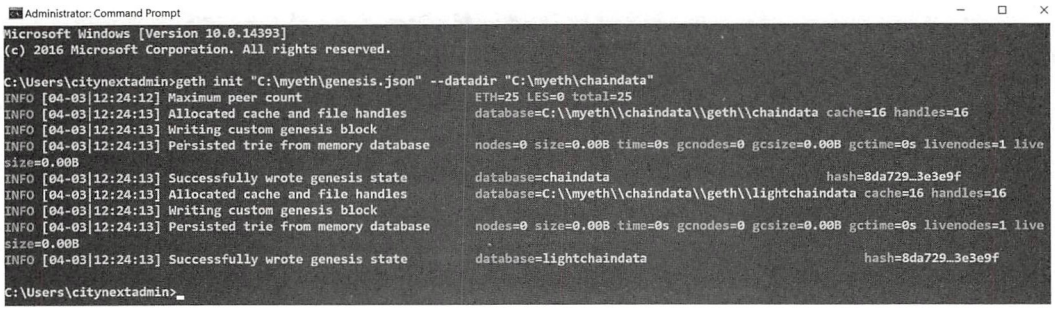
按照下面的步骤，我们开始搭建一个私有网络。

1. 在网络初始化时，Geth 需要使用 genesis.json 文件，同时，需要提供保存区块数据和账户私钥（keystore）的目录。

2. 输入 geth init 命令、genesis.json 文件、存储区块数据和 keystore 的文件夹，进行初始化。

```
C:\Windows\system32>geth init "C:\myeth\genesis.json" --datadir
"C:\myeth\chaindata"
```

输出结果如下：

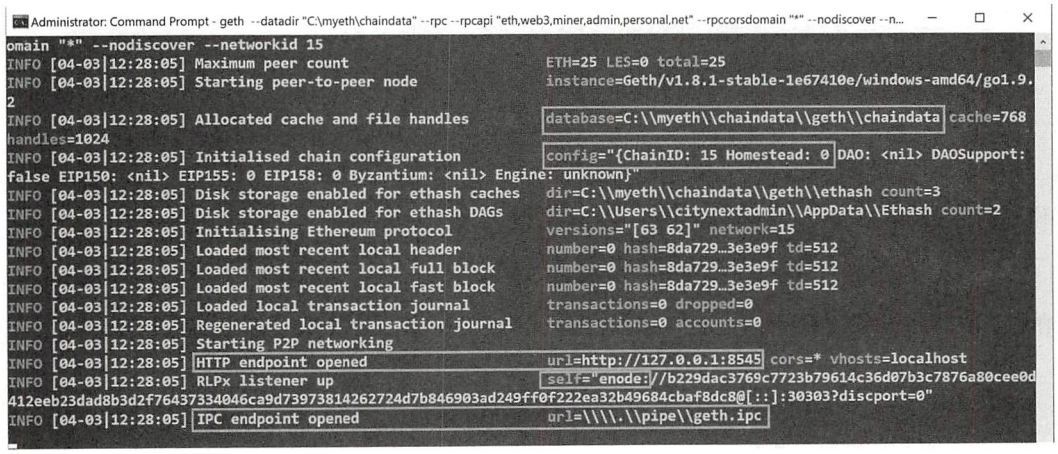


3. 在上面的截图中，我们可以看到创世区块已经产生了，Geth 节点可以启动了。Geth 启动时默认使用 IPC 协议。为了确保 Geth 节点可以通过 RPC 协议访问，命令中需要带上 RPC 参数。

4. 为了搭建节点运行的环境，接下来执行以下命令：

```
geth --datadir "C:\myeth\chaindata" --rpc --rpcapi "eth,web3,miner,
admin,personal,net" --rpccorsdomain "*" --nodiscover --networkid 15
```

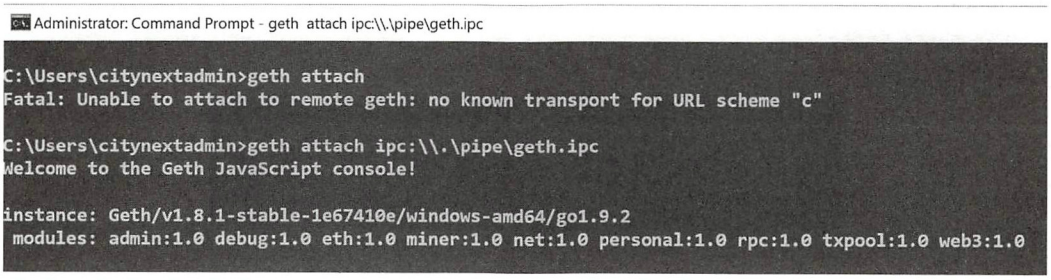
执行结果如下：



在执行命令中，我们可以看出，命令使用了 datadir 参数，启用了 RPC，暴露给外部的模块和 API，Networkid 是 15，说明这是一个私有网络[⊖]。从执行结果上看，我们会发现一些比较有价值的信息。首先，ehterbase 或 coinbase 没有设置，它在挖矿前需要设置好，目前，挖矿任务还没有启动（也可以使用命令行的方式去启动挖矿程序）。屏幕上显示了区块数据存储的位置，也显示了 ChainID，以及是否连接到了 Homestead 主网，数字为 0 表明没有连接到主网。屏幕上的 enode 的信息是节点在网络上的身份标志。如果其他节点准备加入这个网络中，就需要提供自己的 enode 值。最后，屏幕上显示了 IPC 和 RPC 协议正在运行和接受请求。RPC 访问可以通过 http://127.0.0.1:8545 或 http://localhost:854，IPC 访问可以通过 \\.\pipe\geth.ipc 命令如下：

```
geth --datadir "C:\myeth\chaindata" --rpc --rpcapi
"eth,web3,miner,admin,personal,net" --rpccorsdomain "*" --
nodiscover --networkid 15
```

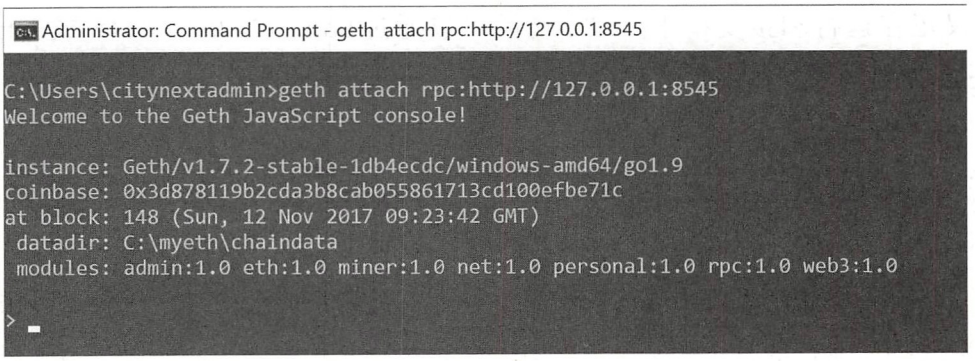
5. 上面的命令将启动以太坊节点，细心的读者可能会发现，命令是以服务的形式持续运行的，这样会导致没有办法再执行其他命令了，所以，为了对这些运行中的节点进行管理，我们可以在本机上打开另外一个命令行窗口，输入 Geth attach ipc:\\.\pipe\geth.ipc 使用 IPC 协议连接到节点上，这样就可以执行其他命令了。如下图显示：



⊖ 前面提到过，ID 大于 4 就是私有网络。——译者注

6. 连接本地的私有网络，可以使用RPC方式接入，输入命令`Geth attach rpc:http://localhost:8545`或`get attach rpc:http://127.0.0.1:8545`。如果你看到的结果和上面显示的不同，那是因为在我这个例子中，coinbase账户已经设置了。后面将介绍coinbase账户。

7. 默认的RPC接入端口是`8545`，可以在命令行中使用`-rpcport`参数进行设置。IP地址也可以通过参数`-rpcaddr`设置。



8. 连接到Geth节点后，接下来就需要设置coinbase或etherbase账户了。首先需要创建一个新账户，创建账户可以使用`personal`对象的`newAccount`方法。创建账户时，需要同时输入密码。下面的屏幕显示了账户创建后的account ID信息。

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xe14d4d757d493d300b11de058f7cba464c0effc8"
```

9. 账户创建完成后，需要确定使用coinbase账户或etherbase账户。如果需要更改原有的coinbase账户地址，可以通`address.miner`对象的`setEtherBase`

函数进行操作。这个操作将使用新的账户替换原有的 coinbase 账户。更改结果将显示为 ture 或 false。

```
> miner.setEtherbase("0xe14d4d757d493d300b11de058f7cba464c0effc8")
true
>
```

10. 执行下面的查询命令，你会发现刚才设置的地址已经生效了。

Coinbase 地址设置成功后，Geth 节点开始运行了，此时，挖矿可以启动了，因为我们只有一个矿工，所以它将获得全部的挖矿奖励，coinbase 账户的以太币会逐步增加。

```
> eth.coinbase
"0xe14d4d757d493d300b11de058f7cba464c0effc8"
> _
```

11. 执行下列代码，启动挖矿：

```
miner.start()
```

也可以用下面的命令

```
miner.start(4)
```

上面的命令执行结果如下：

```
> miner.start(4)
null
> _
```

start 中的参数代表了用于挖矿的线程数量。该命令将启动挖矿程序，同

样可以从屏幕上看到执行结果。

```
INFO [11-13|12:38:31] RLPx listener up                         self=enode://87d6158dc33c3d308bde58c8a107a883a0
fcccc754cbc33513e097d58cebb77bf571e76ff4101793ccecffd8edcd8f40947dfce7d8f52485140bb449f9@[::]:30303
INFO [11-13|12:38:32] IPC endpoint opened: \\.\pipe\geth.ipc
INFO [11-13|12:52:18] Updated mining threads                   threads=4
INFO [11-13|12:52:18] Transaction pool price threshold updated price=18000000000
INFO [11-13|12:52:18] Starting mining operation
INFO [11-13|12:52:18] Commit new mining work                   number=149 txs=0 uncles=0 elapsed=0s
```

12. 如果需要停止挖矿，可以在另外一个窗口中执行`miner.stop()`命令。

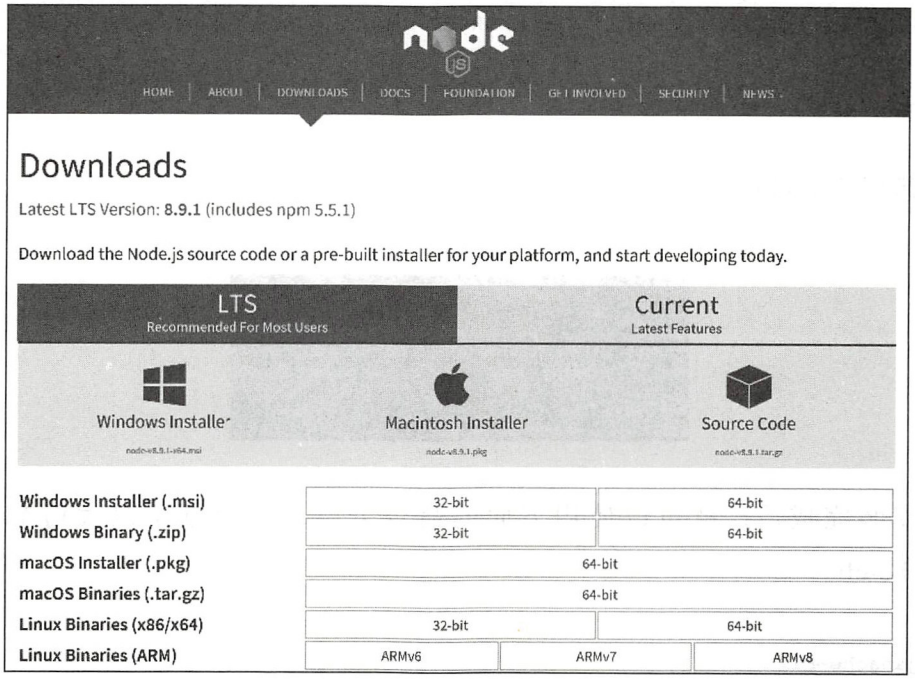
2.4 ganache-cli

在以太坊上，交易写入账本分为两个阶段：

- 创建交易，然后将其放入交易池中。
- 定期从交易池中获取交易，然后开始挖矿。挖矿意味着将这些交易写入以太坊数据库或账本的。

从上面这个过程可以看出，在以太坊上进行解决方案和智能合约的开发和测试，将十分浪费时间。ganache-cli（以前叫作：TestRPC）就是为了缓解这个问题，ganache-cli 包含了以太坊的交易处理流程和挖矿功能，但挖矿时不需要竞赛，交易产生后就会立即写入账本。对于开发者来说，使用 ganache-cli 可以作为以太坊节点，不需要挖矿交易就可以写入账本。

ganache-cli 是基于 Node.js 开发的，因此，在部署 ganache-cli 之前，需要安装 Node.js，可以从网址 https://nodejs.org/en/download/ 上面下载，需要选择相应的 32 位或 64 位操作系统版本，点击下面屏幕上显示的链接就可以下载安装包：



现在，我们下载 64 位的 Windows 安装版本，可以用它来安装 node package manager（NPM）和 Node.js。

下图显示了 Node.js 的版本号是 `v8.9.1`。

```
C:\Users\citynextadmin>node --version
v8.9.1

C:\Users\citynextadmin>_
```

下图显示了 npm 的版本号是 `5.5.1`。

```
C:\Users\citynextadmin>npm --version
5.5.1

C:\Users\citynextadmin>_
```

可以使用 npm install 安装 ganache-cli。

```
npm install -g ganache-cli
```

执行结果如下：

```
C:\Users\citynextadmin>npm install -g ganache-cli
npm WARN deprecated babel-preset-es2015@6.24.1: ??
npm WARN deprecated nomnom@1.8.1: Package no longer
C:\Users\citynextadmin\AppData\Roaming\npm\ganache-
npm WARN webpack-cli@2.0.13 requires a peer of webp

+ ganache-cli@6.1.0
added 492 packages in 70.513s
```

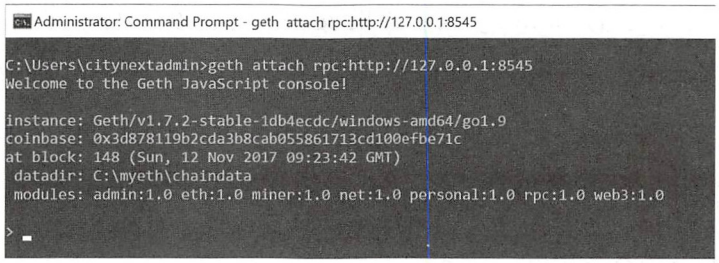
安装完成后，就可以使用下面的命令去启动这个基于以太坊节点的 ganache-cli。

```
ganache-cli
```

上面的命令执行后，默认创建了 10 个账户，每个账户有 100 个以太币，这些账号和私钥如下图所示：

```
C:\Users\citynextadmin>ganache-cli
Ganache CLI v6.1.0 (ganache-core: 2.1.0)

Available Accounts
==================
(0) 0xb2d4fc1bd4971715d4bf67662926b603060fa7e5
(1) 0x400e4d8a1affcec6495bdab627092c139f9c5897
(2) 0x9502159df7cf412132c0b79200ea478b2357dc37
(3) 0xb260a5fe0be2c32667553f836cdc0b52d74befa0
(4) 0x5954c37e2c5dacc2e5a67dc548fd9a393ae70cb7
(5) 0x9f9064d0e6fb2b157db28b6d0424f5087b834ff1
(6) 0xfc8279e0a9976bd51516310b64f23e7459a8cd00
(7) 0xcb4ad724e7e38c1d31770d64578fa50658153219
(8) 0xcce2e065b44d865cbff56a746680bf883f37027a
(9) 0x8fa1b27a7458a5f5cc707763d95cba6d7cfd87d4

Private Keys
==================
(0) 2694d75fa257a63a2e6da1e14ff9b85dcacb7885a659b10e92902541ad433769
(1) e99109f2e419d9243ab319563946b19debfbf39b7808748c65a120a9216597e4
(2) bd448b10725f555f14fbce0b43ce6e04bdd8386926eabb6d1682855b23d52ede
(3) 7c7f645f2aedc3fd89226af07f3b6ae599badf5a74021b377b250f1ad8e3e6a0
(4) 02cc9f83052130e00a95d5f02c6053b9f6c7bc2d40bcb0c454c8cf7f9f48d81c
(5) a8a2984b371d5deb4db984e3ac7dcee7a2ab730a2025aef0cf9ea475c74ec1e7
(6) eadc6697222450bc2cfeeceb999e6ad909fd9916f801ef567c67677fcf0b41d2
(7) f8e486cdff58764bd28d73a855473c29cabf3316076777ef54e858428e64295e
(8) 481186bd532f8d3c9149dbbe8e3afd4daf74034a3d368a17606b38fc6c4e7686
(9) fe5fc2dd37407169bf512622a7ba66a1b70ab54c9c3b6ba56e410c10304dcb6a
```

也可以开启另外一个命令行窗口，用 Geth 命令连接到这个网络，如下图所示：

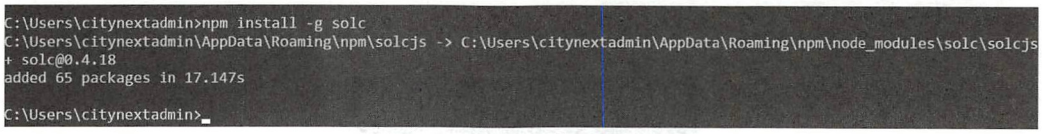


2.5　Solidity 编译器

Solidity 是一种编写智能合约的语言。后面的章节会详细介绍智能合约。编写完 Solidity 代码后，需要使用 Solidity 编译器去编译，它将生成字节码和其他输出物，它们在部署智能合约时会用到。在以前，Solidity 是 Geth 安装包的一部分，现在已经脱离了 Geth 的安装过程，需要自行安装。Solidity 编译器被称之为 `solc`，可以使用 `npm` 命令进行安装：

```
npm install -g solc
```

执行结果，如下图所示：



2.6　web3 JavaScript 库

`web3` 库是开源的 JavaScript 库，可以连接到本地或远程的以太坊节点，它

使用的是 IPC 或 RPC 协议。web3 库是面向客户端的，可以和 Web 页面一起使用，以便向以太坊节点发起查询请求和交易提交请求。我们可以使用节点管理工具来进行安装，就像 Solidity 编译器的安装一样。在编写本书的时候，web3 的最新版本无法运行并提示没有正确安装，这是因为丢失了 BigNumber.js 文件。但是，以前的稳定版本可以连接到以太坊节点的后台。下面的步骤显示了如何安装 web3 JavaScript 库。

1. 使用下面的命令安装 web3

```
npm install web3@0.19
```

执行后，产生的结果如下：

```
C:\Users\citynextadmin>npm install web3@0.19
npm WARN saveError ENOENT: no such file or directory, open 'C:\Users\citynextadmin\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'C:\Users\citynextadmin\package.json'
npm WARN citynextadmin No description
npm WARN citynextadmin No repository field.
npm WARN citynextadmin No README data
npm WARN citynextadmin No license field.

+ web3@0.19.1
updated 1 package in 2.681s

C:\Users\citynextadmin>
```

2.web3 安装后，可以使用 Node.js 调用。在命令行模式下，如下图所示，执行 node 命令，将进入 node 模式。

```
C:\Users\citynextadmin>node
> 
```

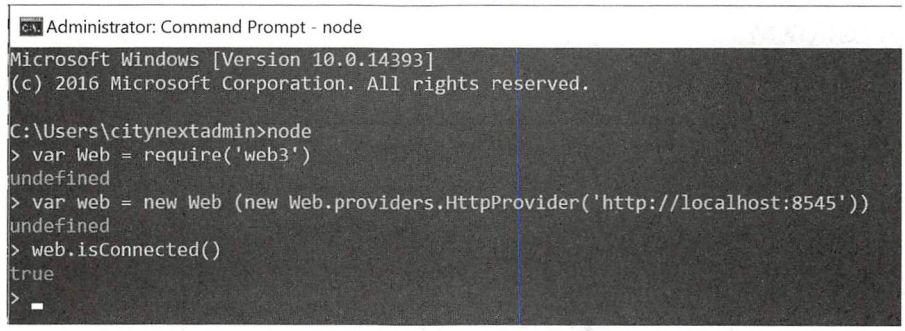
3. 在 node 模式下，输入下面的命令就能连接到以太坊节点。以太节点是 TestRPC 或基于 Geth 的私有网络。web3 可以使用 WebSockets、IPC 或 RPC 协议连接到以太坊节点。下面的例子是 web3 使用 RPC 协议来连接到以太坊节点：

严禁网络传播本PDF，违者责任自负！


```
var Web = require('web3')
var web = new Web (new
Web.providers.HttpProvider('http://localhost:8545'))
```

命令的第一行是加载 web3 模块，第二个命令是新建一个 HttpProvider 实例，用于连接到本地端口是 8545 的以太坊节点。

4. 为了验证 web3 是否连接到了以太坊节点，可以执行 isConnected 方法。如果返回值是 true，那么就说明 web3 已经连接上了，如下图所示：

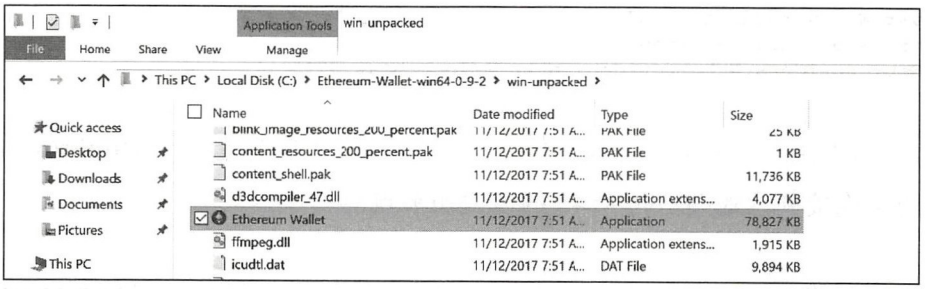


2.7　Mist 钱包

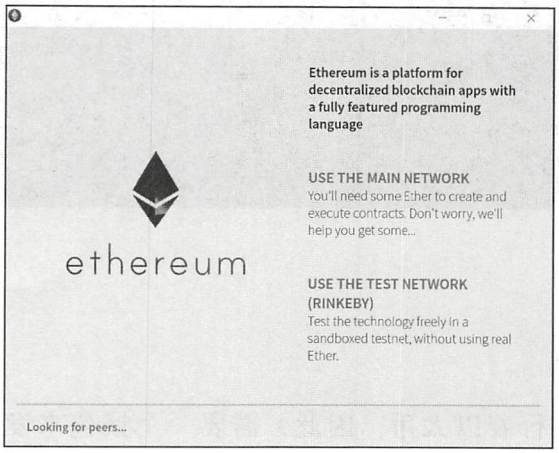
以太坊上面运行着以太币，因此，需要一个钱包来发送和接收以太币。Mist 就是这样一个接收和发送以太币的钱包，同时它也有利于用户在以太网络上（包括公有和私有网络）进行交易。在 Mist 上，用户可以创建账户、发送和接收以太币、部署和调用智能合约。

Mist 可以在 https://github.com/ethereum/mist/releases 网站下载。用户需要根据自己的情况选择一个合适的安装包（在本例中，我们是在 windows 2016 环境下，所以，下载的是 Ethereum-Wallet-win64-0-9-2.zip），下载后解压缩。解压后，文件如下图所示，双击 Ethereum Wallet 图标。

严禁网络传播本PDF，违者责任自负！



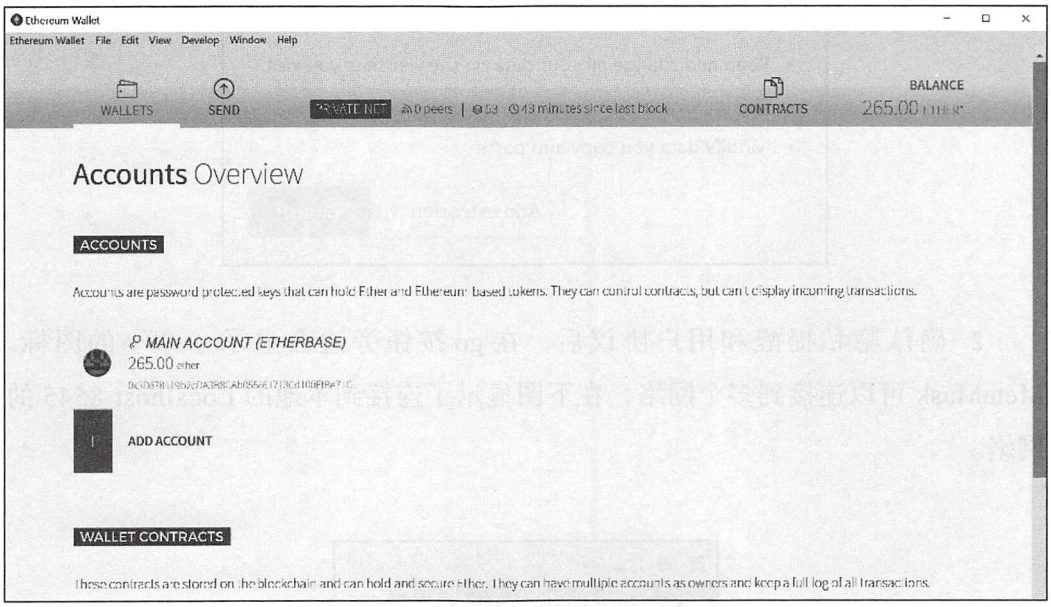
Mist 就会启动。Mist 具有一定的智能性，如果本地运行了一个私有链，它就会识别到并连接上去。如果本地没有任何私有网络，它就会连接到主网或 Rinkeby 测试网络。



在本书中，前面已经搭建了一个私有网络，下图则显示了其连接的情况：

严禁网络传播本PDF，违者责任自负！

一旦连接成功，就可以与以太坊网络进行交互、发送和接收以太币、部署智能合约、调用智能合约里面的函数。

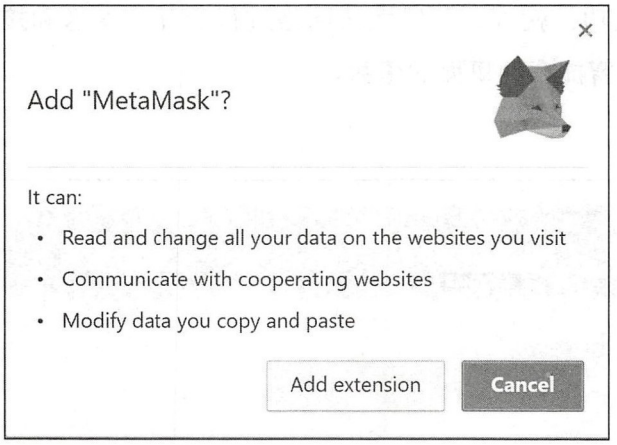


2.8　MetaMask

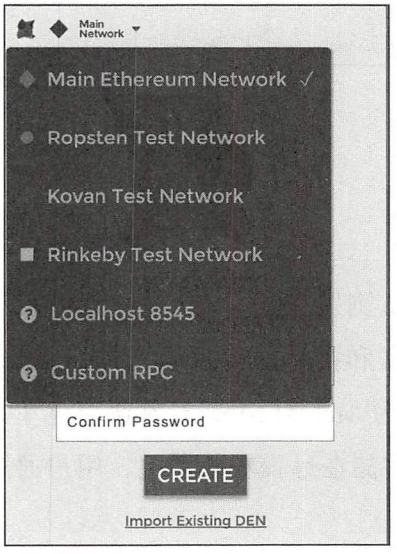
MetaMask 是 Chrome 浏览器的一个轻量级插件，它能够与以太坊网络进行交互。它也是一个发送和接收以太币的钱包。MetaMask 可以从网址 https://metamask.Io 处下载。因为 MetaMask 运行在浏览器里，因此，区块数据没有办法下载到本地，只能是存储在远程服务器上，用户通过浏览器去访问它。请看下面的操作步骤：

1.MetaMask 可以作为插件添加，如下图所示：

严禁网络传播本PDF，违者责任自负！



2. 确认隐私提醒和用户协议后，在 go 按钮旁边会显示一个小的图标。MetaMask 可以连接到多个网络，在下图显示了连接到本地的 Localhost 8545 的网络。

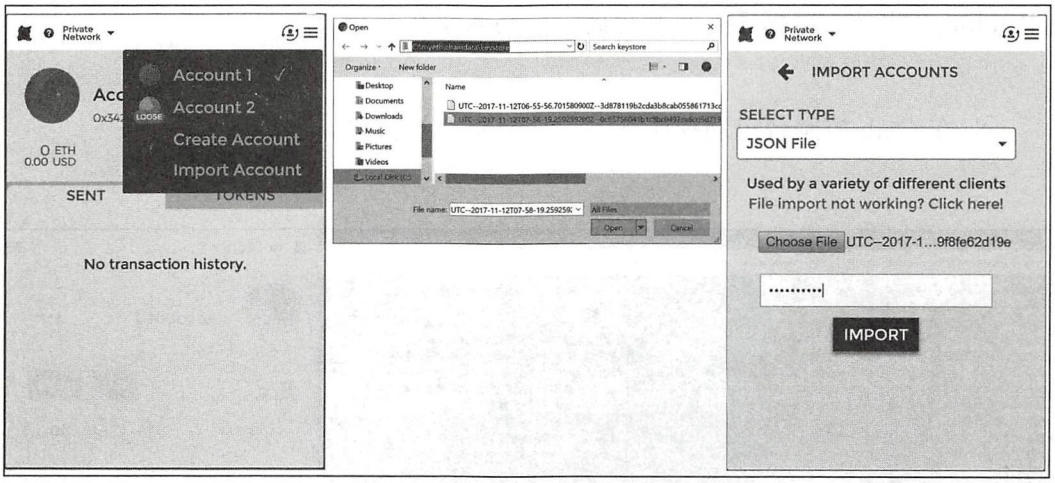


3. 在 MetaMask 中，为了验证身份，在创建一个密钥时，需要提供保护密码。这些信息存储在 MetaMask 服务器的密钥库里，如下图所示：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

4. 点击 Account 图标，使用 Import Account 菜单，可以导入已经存在的账户：

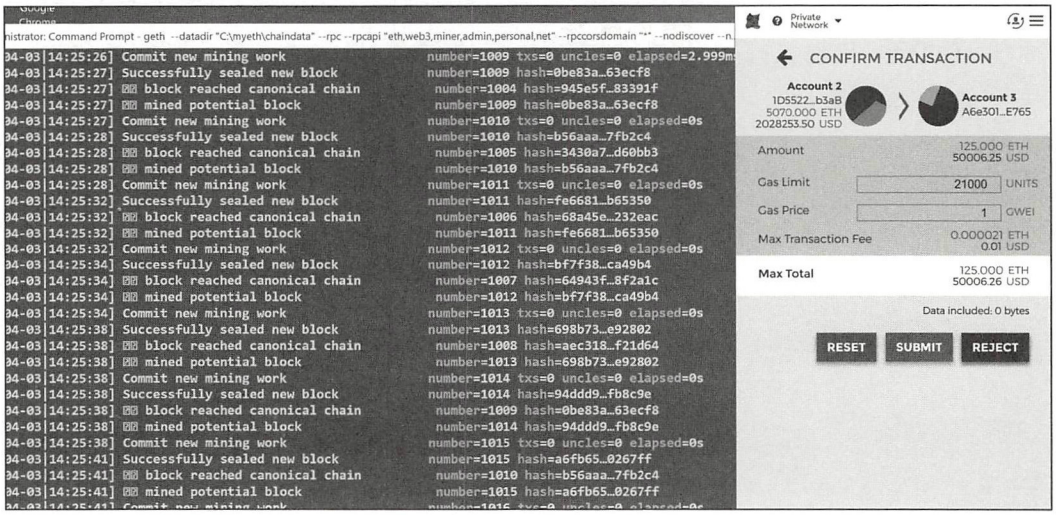


5. 账户创建后，MetaMask 可以通过以太坊的交易从一个账户向另外一个账户转账。

6. 为了向另外一个账户发送以太币，首先选择一个发送账户，然后点击 Send 按钮，在弹出的窗口中，输入目标账户地址、金额，然后点击 Next 按钮：

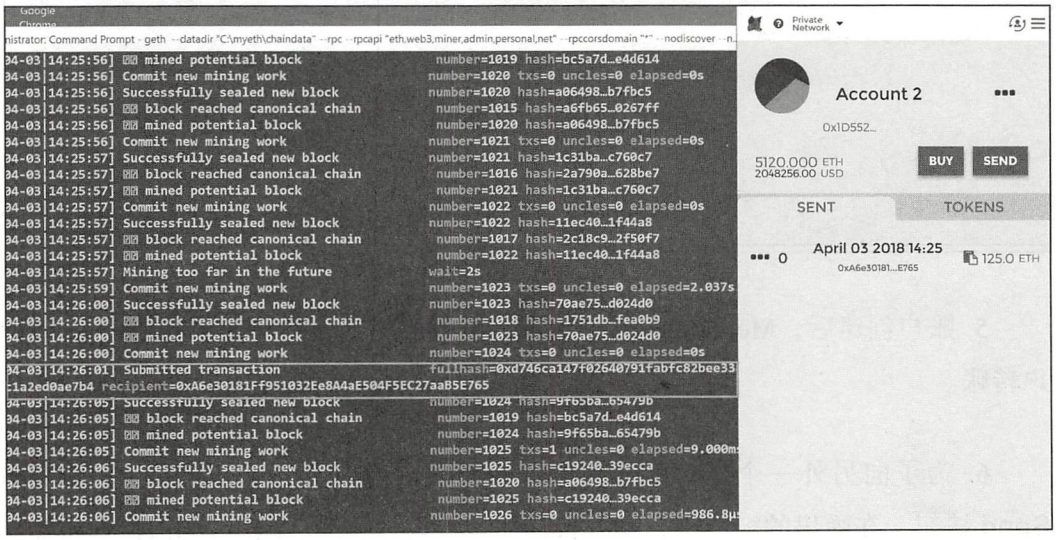
非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！



7. 点击 submit 按钮，提交交易。此时交易将保存在交易池里，处于等待（Pending）状态。挖掘任务开始将此事务写入永久存储器。

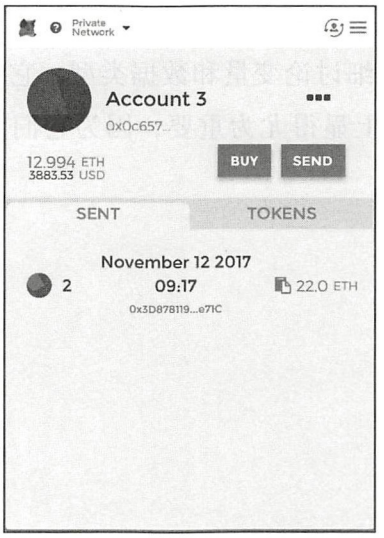
8. 在 Geth 控制台启动挖矿任务，交易将会被打包，如下图所示：



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

过了一会，交易被写入账本，MetaMask 中的双方的账户余额就会发生变化。



2.9　本章小结

本章涉及很多实质性的内容。以太坊节点提供了 JSON RPC 的连接方式，我们可以使用 WebSockets、IPC、RPC 等多种方式进行连接。本章我们介绍了各种区块链网络：公共网、主网、测试网和私有网。本章也介绍了如何构建私有网络，同时，介绍了如何搭建一个开发环境，它会在下一章用到。本章重点介绍了在 windows 环境下如何安装各种工具。这些工具都有各种的用途。很显然，一些工具之间存在功能重叠之处。例如，基于 Geth 的私有网络和 ganache-cli 开发环境都能够创建以太节点，但还是存在一些差异。本章也介绍了如何安装 Geth、Solidity 编译器、ganache-cli、web3 库、JavaScript 框、Mist 和 MetaMask 钱包。一些读者习惯于使用 ganache-cli，而其他一些人则可能喜欢使用基于 Geth 的以太节点。在接下来的章节中，会介绍另外一个重要的工具：Truffle。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

在第 3 章中，我们将重点介绍 Solidity 开发语言，也是本书的主要内容。Solidity 支持面向对象，在多种复杂的数据类型之外，它又提供了原生的函数声明和定义方式，用于接收参数和返回值。它提供了程序控制结构和表达式，以及更多特性。第 3 章将详细讨论变量和数据类型，它们是所有程序语言的核心内容，这一点在 Solidity 上显得尤为重要，因为它们要将数据保存在分布式账本上。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

第 3 章 Chapter 3

Solidity 介绍

从这章开始，我们将学习 Solidity 语言。前两章介绍了区块链、以太坊和它们的工具集。此外，还讨论了一些与区块链相关的重要概念，这些概念对于更好地理解和编写 Solidity 中的高效代码是必要的。有多种编程语言支持 EVM。其中有些语言已被弃用，另一些则被不同程度地认可。Solidity 是目前在 EVM 上最流行的语言。从本章开始，将重点介绍 Solidity 及其概念，以及如何编写高效的智能合约。

3.1 以太坊虚拟机

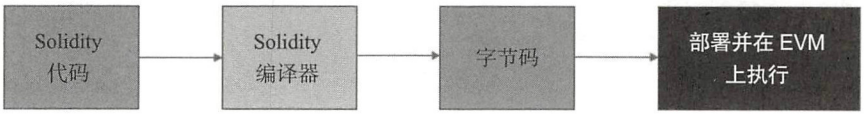
Solidity 是针对以太坊虚拟机（EVM）的编程语言。以太坊区块链通过编写和执行称为智能合约的代码来帮助扩展其功能。我们将在后面的章节中深入讨论智能合约的细节，但就目前而言，知道智能合约与 Java 或 C ++ 中面向对象的类相似就足够了。

非卖品，仅供非商业用途或交流学习使用

EVM 执行作为智能合约的一部分的代码。智能合约是用 Solidity 语言写的，然而，EVM 并不理解 Solidity 的高级结构。EVM 可以理解的是称为**字节码**的低级指令。

需要编译器将 Solidity 代码转换为 EVM 可理解的字节码。Solidity 附带的编译器称为 Solidity 编译器或 solc，它负责完成这种转换。在上一章，我们使用 Node.js npm 命令下载并安装了 Solidity 编译器。

整个过程如下图所示，从在 Solidity 中编写代码到在 EVM 中执行代码：



在上一章，当我们编写 `HelloWorld` 合约的时候，已经探讨过我们的第一个 Solidity 代码了。

3.2 Solidity 和 Solidity 文件

Solidity 是一种非常接近 JavaScript 的编程语言。在 Solidity 中可以找到 JavaScript 和 C 之间的相似之处。Solidity 是一种静态类型、区分大小写的面向对象编程（OOP）语言。虽然它是面向对象的，但支持有限的面向对象特征。这意味着在编译时，应该定义并且已知变量的数据类型。应该按照面向对象编程的方式来定义函数和变量。在 Solidity 中，Cat 不同于 CAT、cat 或任何其他 cat 的变体。Solidity 语句的终结符是分号（;）。

在扩展名为 `.sol` 的 Solidity 文件中编写 Solidity 代码。Solidity 文件是人类可读的文本文件，可以在任何编辑器（包括记事本）中打开。

非卖品，仅供非商业用途或交流学习使用

Solidity 文件由以下四个高级结构组成：

- 预编译指令
- 注释
- 导入
- 合约 / 库 / 接口

3.2.1　预编译指令

预编译指令通常是任何 Solidity 文件中的第一行代码。pragma 是指定当前 Solidity 文件编译器版本的指令。

Solidity 是一种新的语言，并且在持续改进。每当引入新特性或改进时，都会推出新的版本。本书写作时的版本是 0.4.19。

在 pragma 指令的帮助下，你可以针对你的代码选择相应的编译器版本，如下面代码所示：

```
pragma Solidity ^0.4.19;
```

虽然不是强制性的，但将 pragma 指令作为 Solidity 文件中的第一条语句是一种很好的做法。

pragma 指令的语法如下：

```
pragma Solidity <<version number>> ;
```

需要注意指令是区分大小写的，随后的有效的版本号和语句以分号结束。

版本号由两个数字组成 ——**主版本号**和**次版本号**。

非卖品，仅供非商业用途或交流学习使用

上例中的主版本号是 4，次版本号是 19。通常，在次版本中很少或没有重大更改，但主版本之间可能会有重大更改。你应该选择最满足需求的版本。

字符也被称为**脱字符号**，在版本号中是可选的，但在根据以下规则决定版本号方面起着重要作用：

- ^ 字符是指主版本中的最新版本。因此，^0.4.0 是指版本号为 4 的最新版本，当前为 0.4.19。
- ^ 字符除了提供的主版本号之外不针对任何其他主版本。
- Solidity 文件只能用主版本号为 4 的编译器进行编译，不能在其他主版本号的编译器上编译。

使用精确的编译器版本而不是使用 ^ 来指定编译 Solidity 代码，是一种更好的做法。当在 `pragma` 指令中使用 ^ 字符时，如果编译器的新版本有改动，可能会弃用你的代码。例如，`throw` 语句已被弃用，并建议在较新版本中使用较新的结构，如 `assert`、`require` 和 `revert`。你不会想在某一天惊讶地发现你的代码表现得和之前不同。

3.2.2 注释

任何编程语言都提供了注释代码的功能，Solidity 也是如此。Solidity 有以下三种注释方式：

- 单行注释
- 多行注释
- 以太坊自然规范（Natspec）

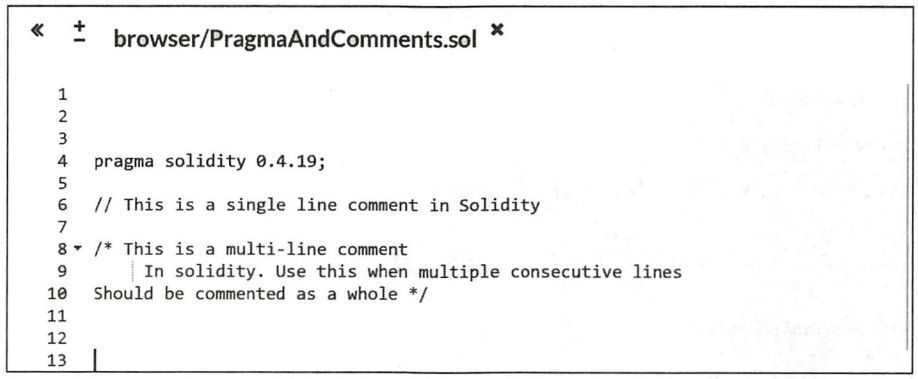
单行注释用双斜线表示 `//`，多行注释用 `/*` 和 `*/` 表示。Natspec 有两种格式：`///` 用于单行，`/**` 和 `*/` 分别用于多行注释的开始和结尾。Natspec

用于文件，它有自己的规范。可从网址 https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format 获取整个规范。

我们通过以下代码看一下 Solidity 的注释：

```
// 这是 Solidity 中的单行注释
    /* 这是 Solidity 中的多行注释。
多个连续行作为整体来注释 */
```

在 Remix 中，pragma 指令和注释如下图所示：



3.2.3　import 语句

import 关键字帮助我们导入其他 Solidity 文件，以便在当前的 Solidity 文件和代码中访问其代码。这有助于我们编写模块化的 Solidity 代码。使用 import 的语法如下所示：

```
import <<filename>> ;
```

文件名可以是完全显式或隐式路径。斜杠“ / ”用于将目录从其他目录和文件中分离出来，“ . ”用于引用当前目录，“ .. ”用于引用父目录。这与 Linux

bash 引用文件的方式非常相似。这里显示了一个典型的 `import` 语句。另外，请注意以下代码结尾的分号：

```
import 'CommonLibrary.sol';
```

3.2.4 合约

除了 `pragma`、`import` 和注释外，我们还可以定义全局或顶级的合约、库和接口。我们将在随后的章节中深入探讨合约、库和接口。本章假定你知道可以在同一个 Solidity 文件中声明多个合约、库和接口。以下屏幕截图中显示的 `library`、`contract`、和 `interface` 关键字区分大小写：

```
//contracts.sol

pragma solidity 0.4.19;

// This is a single line comment in Solidity

/* This is a multi-line comment
    In solidity. Use this when multiple consecutive lines
Should be commented as a whole */

contract firstContract {

}

contract secondContract {

}

library stringLibrary {

}

library mathLibrary {

}

interface IBank{

}

interface IAccount {

}
```

3.3　合约的结构

Solidity 的主要目的是为以太坊编写智能合约。智能合约是 EVM 部署和执行的基本单元。尽管本书后面的多个章节专门用于编写和开发智能合约，但本章将讨论智能合约的基本结构。

从技术上讲，智能合约由两个结构——变量和函数组成。变量和函数都有多个方面，这也是本书将讨论的内容。本节主要介绍使用 Solidity 语言的智能合约的常规结构。

合约由以下多个结构组成：

- 状态变量
- 结构定义
- 修改器定义
- 事件声明
- 枚举定义
- 函数定义

典型的合约由前面提到的所有结构组成。需要注意，在下面的截图中，这些结构中的每一个由多个其他结构组成，在后续章节详细讨论这些主题时再进行相应的讨论：

```
pragma solidity 0.4.19;

//contract definition
contract generalStructure {
    //state variables
    int public stateIntVariable; // vriable of integer type
    string stateStringVariable; //variable of string type
    address personIdentifier; // variable of address type
    myStruct human; // variable of structure type
    bool constant hasIncome = true; //variable of constant nature
```

```
    //structure definition
    struct myStruct {
        string name; //variable fo type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
    }

    //modifier declaration
    modifier onlyBy(){
        if (msg.sender == personIdentifier) {
            _;
        }
    }

    // event declaration
    event ageRead(address, int );

    //enumeration declaration
    enum gender {male, female}

    //function definition
    function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

       human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

       gender _gender = gender.male; //using enum

       ageRead(personIdentifier, stateIntVariable);
    }

}
```

3.3.1 状态变量

编程中的变量是指可以包含值的存储单元。这些值可以在运行时更改。可以在代码中的多个位置使用变量，并且它们都将引用存储的值。Solidity 提供了两种类型的变量——状态变量和内存变量。在本节中，我们将介绍状态变量。

状态变量是 Solidity 合约最重要的特性之一。状态变量由矿工永久存储在区块链 / 以太坊账本中。在合约中，没有在任何函数内声明的变量称为**状态变量**。状态变量存储合约的当前值。状态变量的内存是静态分配的，并且在合约生命周期内不能更改（分配的内存大小）。必须静态定义每个状态变量的类型。Solidity 编译器必须确定每个状态变量的内存分配细节，因此必须声明状态变量的数据类型。

状态变量还有相关联的限定符，可以是以下任何一种：

- `internal`：默认情况下，如果没有指定任何内容，则状态变量具有 `internal` 限定符。这意味着这个变量只能在当前的合约函数和任何继承它们的合约中使用。这些变量不能被外部访问修改，但是，可以查看它们。内部状态变量的示例如下所示：

```
int internal StateVariable ;
```

- `private`：这个限定符就像 `internal` 上附加的约束。私有状态变量只能在声明它们的合约中使用。即使在派生合约中也不能使用它们。私有状态变量的示例如下所示：

```
int private privateStateVariable ;
```

- `public`：这个限定符使得状态变量可以直接访问。Solidity 编译器为每个公共状态变量生成一个 `getter` 函数。公共状态变量的示例如下：

```
int public stateIntVariable ;
```

- `constant`：这个限定符使得状态变量不可变。变量声明时必须赋初值。实际上，编译器会在所有代码中将变量的引用替换为指定的值。常量状态变量的示例如下：

```
bool constant hasIncome = true;
```

如前所述，每个状态变量都有一个关联的数据类型。数据类型帮助我们确定变量的内存需求并确定可以存储在其中的值。例如，类型为 `uint8` 的状态变量称为**无符号整数**，它将被分配一个预定的内存大小，并且值的范围为 0 ～ 255。任何其他值都被认为是外部的，编译器和运行时不接受将其存储在变量中。

Solidity 提供了以下多种开箱数据类型：

- `bool`

- uint/int
- bytes
- address
- mapping
- enum
- struct
- bytes/String

使用 enum 和 struct，可以声明自定义的用户数据类型。在本章后面，将详细介绍数据类型和变量。

3.3.2 结构

结构或结构体有助于实现自定义的用户数据类型。结构是一种复合数据类型，由多个不同数据类型的变量组成。它们与合约非常相似，但是，它们不包含任何代码。它们只包含变量。

有时你想将相关数据存储在一起。假设你想存储关于员工的信息，比如员工姓名、年龄、婚姻状况和银行账号。为了表示这些与单个员工相关的变量，可以使用 Solidity 的结构中的 struct 关键字进行声明。结构中的变量在花括号 {} 内定义，如下截图所示：

```
//structure definition
struct myStruct {
    string name; //variable fo type string
    uint myAge; // variable of unsigned integer type
    bool isMarried; // variable of boolean type
    uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
}
```

使用下面的语法来创建一个结构的实例。不需要显式调用关键字 new。关

键字 new 只用于创建合约或者数组的实例，如下截图所示：

```
human = myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct
```

在函数中可以创建多个结构体的实例。结构体可以包含数组和映射变量，映射和数组可以存储结构体的值。

3.3.3　修改器

在 Solidity 中，修改器总是与函数关联。编程语言中的修改器是指改变执行代码行为的结构。由于修改器与 Solidity 中的某个函数相关联，因此它可以改变与其关联的函数的行为。为了便于理解修改器，可以将其视为在执行目标函数之前执行的函数。假设你想调用 getAge 函数，但是在执行它之前，你想要执行另一个函数来检查合约的当前状态、传入参数中的值、状态变量中的当前值等，并相应地决定是否应该执行目标函数 getAge。这有助于编写更整洁的函数，而不会用验证和确认规则混淆它们。此外，修改器可以与多个函数关联。这确保了更清晰、更易读、更易维护的代码。

修改器使用 modifier 关键字后跟修改器标识符、它应该接受的任何参数、花括号 {} 内编写的代码来进行定义。修改器中的“_”（下划线）表示执行目标函数。你可以将此视为下划线被内联的目标函数替换。payable 是一种由 Solidity 提供的开箱即用的修改器，当应用于任何函数时允许该函数接受以太币。

在合约级别声明了 modifier 关键字，如以下截图所示：

```
//modifier declaration
modifier onlyBy(){
    if (msg.sender == personIdentifier) {
        _;
    }
}
```

正如我们所看到的，在代码片段前面的截图中，名为 `onlyBy()` 的修改器在合约级别被声明。它使用 `msg.sender` 检查传入地址的值，`msg.sender` 保存地址并将其存储在状态变量中。如 `msg.sender` 对读者来说可能还无法理解，我们将在下一章深入介绍这些内容。

该修改器与 `getAge` 函数关联，如以下截图所示：

```
//function definition
function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

   human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

   gender _gender = gender.male; //using enum
}
```

`getAge` 函数只能由与合约的 `_personIdentifier` 状态变量中存储的地址相同的账户执行。如果任何其他账户尝试调用该函数，该函数将不会执行。

需要注意的是，任何人都可以调用 `getAge` 函数，但执行只会发生在单个账户中。

3.3.4 事件

Solidity 支持事件。Solidity 中的事件就像其他编程语言中的事件一样。事件是从合约中触发的，任何对它们感兴趣的人都可以捕获它们并响应执行代码。Solidity 中的事件主要用于通过 EVM 的日志工具向调用应用程序通知合约的当前状态。它们用于通知应用程序有关合约中的改变，并且应用程序可以使用它们来执行相关逻辑。它们不是应用程序，而是轮询合约中特定状态的更改，合约可以通过事件通知它们。

合约中声明的事件在全局域有效，并且被合约中的函数所调用。使用

严禁网络传播本PDF，违者责任自负！

event 关键字声明一个事件，后跟一个标识符和参数列表并以分号结尾。参数中的值可用于记录信息或执行条件逻辑。事件信息及其值作为交易的一部分存储在区块内。在上一章讨论交易的属性时，引入了一个名为 LogsBloom 的属性。作为交易的一部分引发的事件存储在此属性中。

没有必要显式地提供参数变量——只有数据类型就足够了，如以下截图所示：

```
// event declaration
event ageRead(address, int );
```

可以从任何函数调用事件的名称并传递所需的参数，如下面的截图所示：

```
//function definition
function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

   human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

   gender _gender = gender.male; //using enum

   ageRead(personIdentifier, stateIntVariable);
}
```

3.3.5　枚举

enum 关键字用于声明枚举。枚举用于在 Solidity 中声明用户自定义的数据类型。枚举包含一个枚举列表、一组预定义的常量。

在 Solidity 中，枚举中的常量值可以显式地转换为整数。每个常量值对应一个整数值，第一个值为 0，每个连续项的值增加 1。

使用 enum 关键字声明枚举，后面跟着枚举标识符和花括号 {} 内的枚举值列表。值得注意的是，枚举声明没有分号作为终结符，并且列表中至少应该声

严禁网络传播本PDF，违者责任自负！

明一个成员。

枚举的示例如下：

```
enum gender {male, female}
```

可以声明并赋值枚举变量，如下面的代码所示：

```
gender _gender = gender.male ;
```

在 Solidity 合约中并没有强制要求定义枚举。如果如前面的例子所示有一个不变的项目常量列表，则应该定义枚举。这是声明枚举的很好例子。它们使你的代码更具可读性和可维护性。

3.3.6 函数

函数是以太坊和 Solidity 的核心。以太坊维护状态变量的当前状态，并执行交易以更改状态变量中的值。当调用或触发合约中的某个函数时，会导致创建一个交易。函数机制是为了从状态变量读取值和向状态变量写入值。函数是一个按需调用执行的代码单元。函数可以接受参数，执行其逻辑，并可选地将值返回给调用者。可以以匿名方式命名函数。Solidity 提供了命名函数，在合约中只能有一个称为 fallback 函数的未命名函数。本书后面我们将更多地了解 fallback 函数。

在 Solidity 中声明函数的语法如下：

```
//function definition
function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {


}
```

严禁网络传播本PDF，违者责任自负！

使用 function 关键字后跟其标识符 getAge 来声明函数。它可以接受多个以逗号分隔的参数。参数标识符是可选的，但应该在参数列表中提供数据类型。函数可以附加修改器，比如 onlyBy()。

有几个额外的限定符会影响函数的行为和运行。函数具有可见性限定符以及与可在函数中执行的操作有关的限定符。下面讨论可见性和与函数能力相关的关键字。函数也可以返回数据，并且使用 return 关键字来声明这些信息，然后是返回参数列表。Solidity 可以返回多个参数。

函数具有与状态变量类似的可见性限定符。函数的可见性可以是以下任何一种：

- public：这种可见性使得函数可以直接从外部访问。它们成为合约接口的一部分，可以在内部和外部调用。
- internal：默认情况下，如果没有指定，则状态变量具有 internal 限定符。这意味着此函数只能用于当前合约以及任何从其继承的合约。这些函数不能从外部访问，它们不是合约接口的一部分。
- private：私有函数只能在声明它们的合约中使用，即使在派生合约中也不能使用它们。它们不是合约接口的一部分。
- external：这种可见性使得函数可以直接从外部但不是内部访问。这些函数是合约接口的一部分。

函数还可以具有以下附加限定符，这些限定符能够更改合约状态变量：

- constant：这些函数不具有修改区块链状态的能力。它们可以读取状态变量并返回给调用者，但不能修改任何变量、触发事件、创建另一个合约、调用其他可以改变状态的函数等。将常函数看作可以读取和返回当前状态变量值的函数。

严禁网络传播本PDF，违者责任自负！

- `view`：这些函数是常量函数的别名。
- `pure`：`pure` 函数进一步限制了函数的能力。`pure` 函数既不能读取也不能写入，即它们不能访问状态变量。使用此限定符声明的函数应确保它们不会访问当前状态和交易变量。
- `payable`：使用 `payable` 关键字声明的函数能够接受来自调用者的以太币。如果发送者没有提供以太币，则调用将会失败。如果一个函数被标记为 `payable`，该函数只能接受以太币。

我们将在后面的章节中详细讨论前面的限定符。可以通过名字来调用函数。

3.4 Solidity 中的数据类型

Solidity 数据类型可以大致分为以下两种类型：

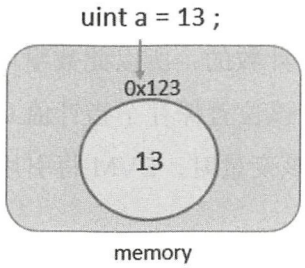
- 值类型
- 引用类型

这两种类型在变量赋值和存储在 EVM 中的方式方面有所不同。可以通过创建一个新副本或者仅仅通过处理引用来完成变量的赋值。值类型维护变量的独立副本，并且在一个变量中更改值不会影响另一个变量中的值。但是，更改引用类型变量中的值可确保任何引用该变量的地方都会获取更新值。

3.4.1 值类型

如果一个类型将数据（值）直接保存在内存中，则称该类型为值类型。值类型在自己的存储空间中保存数值。下图说明了这一点。在此示例中，数据类型为**无符号整数**（uint）的变量声明 13 作为其数据（值）。变量 a 具有由 EVM 分配的存储空间 0x123，并且该位置具有存储的值 13。访问这个变量将直接得到值 13：

严禁网络传播本PDF，违者责任自负！



值类型是大小不超过 32 字节内存的类型。Solidity 提供以下值类型：

- `bool`：可以保存 true 或 false 作为其值的布尔值
- `uint`：这是无符号整数，只能保存 0 和正值
- `int`：这是可以保存负值和正值的有符号整数
- `address`：这表示以太坊环境中的账户地址
- `byte`：这表示固定大小的字节数组（`byte1` 到 `bytes32`）
- `enum`：可以保存预定义的常量值的枚举

值传递

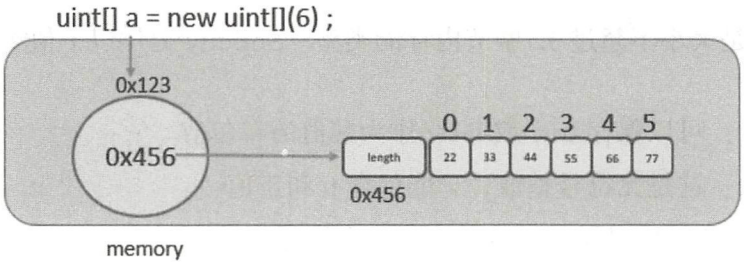
如果将值类型变量赋给另一个变量，或者将值类型变量作为参数传送给函数，则 EVM 会创建一个新变量实例并将原始值类型的值复制到目标变量中。这被称为值传递。更改原始或目标变量中的值不会影响另一个变量中的值。这两个变量将保持其独立的值，并且它们可以在其他变量不知道的情况下更改值。

3.4.2　引用类型

与值类型不同，引用类型不直接将其值存储在变量本身中。它们存储的不是值，而是值存储位置的地址。该变量保存了指向另一个实际存储数据的内存位置的指针。引用类型可以占用大于 32 字节的内存。下面通过图示显示引用类型。

严禁网络传播本PDF，违者责任自负！

在以下示例中，声明了一个数据类型为 uint 的大小为 6 的数组变量。Solidity 中的数组是从 0 开始计数的，所以此数组可以包含 7 个元素。变量 a 由 EVM 分配存储空间 0x123，该位置保存了指针值 0x456。该指针指向存储数组数据的实际内存位置。访问该变量时，EVM 将引用该指针的值并显示数组索引中的值，如下图所示：



Solidity 提供以下引用类型：

- **数组**：这是固定大小或动态大小的数组。细节在本章后面给出。
- **结构**：这是自定义的即用户定义的结构。
- **字符串**：这是字符序列。在 Solidity 中，字符串最终被存储为字节。细节在本章后面给出。
- **映射**：与存储键值对的其他语言中的散列表或字典相似。

引用传递

当引用类型变量被赋给另一个变量时，或者当引用类型变量作为参数传送给函数时，EVM 会创建一个新变量实例并将指针从原始变量复制到目标变量中。这被称为引用传递。这两个变量都指向相同的地址位置。改变原始或目标变量中的值也会改变其他变量的值。这两个变量将共享相同的值，并且一个变量的变化反映在另一个变量中。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

3.5　存储和内存数据位置

在合约中声明和使用的每个变量都有一个数据位置。EVM 提供以下四种用于存储变量的数据结构：

- **存储**：这是可以被合约内所有函数访问的全局内存变量。是以太坊将其存储在环境中每个节点上的永久存储。
- **内存**：合约中的每个函数都可以访问的本地内存。它是生命周期的短暂的内存，当函数执行完成后会被销毁。
- **调用数据**：存储所有传入的函数执行数据，包括函数参数。这是一个不可修改的内存位置。
- **堆栈**：EVM 维护用于加载变量和使用以太坊指令集的变量和中间值的堆栈。这是 EVM 的工作集内存。在 EVM 中，堆栈的深度为 1 024 层，任何超过此数量的深度都会引发异常。

变量的数据位置取决于以下两个因素：

- 变量声明的位置
- 变量的数据类型

基于上述两个因素，有规则规定了如何管理和决定变量的数据位置。数据位置也会影响赋值运算符的工作方式。赋值和数据位置的管理是由规则来解释的。

3.5.1　规则 1

状态变量始终存储在存储数据位置。

3.5.2　规则 2

声明为函数参数的变量始终存储在内存数据位置。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

3.5.3 规则 3

在默认情况下，在函数中声明的变量存储在内存数据位置。但是，有以下几点警告：

- 函数中的值类型变量的存储位置是内存，而引用类型变量的缺省位置是存储。

请注意，在函数内声明的引用类型变量默认保存在存储中。但是，它可以被覆盖。

- 通过覆盖默认位置，引用类型变量可以位于内存数据位置。引用的类型是数组、结构体和字符串。
- 在函数中声明的引用类型不会被覆盖，应该始终指向一个状态变量。
- 在函数中声明的值类型变量不能被覆盖，也不能存储在存储位置。
- 映射总是在存储位置声明，这意味着不能在函数内声明它们。映射不能被声明为内存类型。但是，函数中的映射可以引用声明为状态变量的映射。

3.5.4 规则 4

调用者提供的函数参数始终存储在调用数据位置中。

3.5.5 规则 5

状态变量，通过另一个状态变量赋值，会创建一个新副本。声明了两个值类型状态变量 `stateVar1` 和 `stateVar2`。在 `getUInt` 函数中，`stateVar2` 被赋值给 `stateVar1`。在这个阶段，这两个变量的值都是 `40`。下一行代码将 `stateVar2` 的值更改为 `50` 并返回 `stateVar1`。返回值为 `40`，

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

说明每个变量保持其自己的独立值，如下图所示：

```
pragma solidity 0.4.19;

contract DemoStoragetoStorageValueTypeAssignment {

    uint stateVar1 = 20;

    uint stateVar2 = 40;

    function getUInt() returns (uint)
    {
        stateVar1 = stateVar2;

        stateVar2 = 50;

        return stateVar1; // returns 40

    }
}
```

声明了两个数组类型的状态变量 stateArray1 和 stateArray2。在 getUInt 函数中，stateArray2 被赋值给 stateArray1。在这个阶段，两个变量的值是相同的。下一行代码将 stateArray2 中的一个值更改为 5，并返回 stateArray1 数组同一位置的值。返回值为 4，说明每个变量保持各自的独立值，如以下截图所示：

```
pragma solidity 0.4.19;

contract DemoStoragetoStorageReferenceTypeAssignment {

    uint[2] stateArray1 = [uint(1), 2];

    uint[2] stateArray2 = [uint(3), 4];

    function getUInt() returns (uint)
    {
      stateArray1 = stateArray2;

      stateArray2[1] = 5;

      return stateArray1[1]; // returns 4

    }
}
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

3.5.6 规则 6

将内存变量的值赋给存储变量时总是会创建一个新副本。

声明一个无符号数固定数组状态变量 stateArray。在 getUInt 函数中定义并初始化位于本地内存的固定数组 localArray。下一行代码将 localArray 赋值给 stateArray。在这个阶段，两个变量的值是相同的。下一行代码将 localArray 中的一个值更改为 10，并返回 stateArray1 数组相同位置的元素。返回值为 2，说明每个变量保持各自的独立值，如以下截图所示：

```
pragma solidity 0.4.19;

contract DemoMemorytoStorageReferenceTypeAssignment {

    uint[2] stateArray ;
    function getUInt() returns (uint)
    {
      uint[2] memory localArray = [uint(1), 2];

      stateArray = localArray;

      localArray[1] = 10;

      return stateArray[1]; // returns 2

    }

}
```

声明一个值类型的状态变量 stateVar 并初始化值为 20。在 getUInt 函数中，声明局部变量 localVar 且值为 40。在下一行代码中，将局部变量 localVar 的值赋给 stateVar。在这个阶段，这两个变量的值都是 40。下一行代码将 localVar 的值更改为 50 并返回 stateVar。返回值为 40，说明每个变量保持各自的独立值，如以下截图所示：

非卖品，仅供非商业用途或交流学习使用

```
pragma solidity 0.4.19;

contract DemoMemorytoStorageValueTypeAssignment {

    uint stateVar = 20;
    function getUInt() returns (uint)
    {
      uint localVar = 40;

      stateVar = localVar;

      localVar = 50;

      return stateVar; // returns 40

    }
}
```

3.5.7　规则 7

将状态变量的值赋给内存变量时始终创建一个新副本。声明一个值类型的状态变量 stateVar 并初始化为值 20。在 getUInt 函数中，声明类型为 uint 的局部变量并初始化为 40。将变量 stateVar 的值赋给变量 localVar。在这个阶段，这两个变量的值都是 20。下一行代码将 stateVar 的值更改为 50 并返回 localVar。返回的值是 20，说明每个变量保持各自的独立值，如以下截图所示：

```
pragma solidity 0.4.19;

contract DemoStoragetoMemoryValueTypeAssignment {

    uint stateVar = 20;
    function getUInt() returns (uint)
    {
      uint localVar = 40;

      localVar = stateVar;

      stateVar = 50;

      return localVar; // returns 20

    }
}
```

非卖品，仅供非商业用途或交流学习使用

声明一个固定的状态变量数组 stateArray。在 getUInt 函数中，定义位于本地内存的固定数组 localArray 并使用 stateArray 变量的值进行初始化。在这个阶段，两个变量的值是相同的。下一行代码将 stateArray 中的一个值更改为 5，并返回 localArray1 数组中相同位置的元素。返回值为 2，说明每个变量保持各自的独立值，如以下截图所示：

```
pragma solidity 0.4.19;

contract DemoStoragetoMemoryReferenceTypeAssignment {

    uint[2] stateArray = [uint(1), 2];
    function getUInt() returns (uint)
    {
      uint[2] memory localArray = stateArray;

      stateArray[1] = 5;

      return localArray[1]; // returns 2

    }
}
```

3.5.8 规则 8

将内存变量赋给内存变量不会创建副本；但是，确实为值类型创建了一个新副本。以下截图中显示的代码清单说明了内存中的值类型变量是按值复制的。localVar1 的值不受变量 localVar2 的值更改影响：

```
pragma solidity 0.4.19;

contract DemoMemorytoMemoryValueTypeAssignment {


    function getUInt() returns (uint)
    {
      uint localVar1 = 40;

      uint localVar2 = 80;

      localVar1 = localVar2;

      localVar2 = 100;

      return localVar1; // returns 80

    }
}
```

非卖品，仅供非商业用途或交流学习使用

以下截图中显示的代码清单说明了内存中的引用类型变量通过引用被复制。otherVar 的值受 someVar 变量的影响：

```
pragma solidity 0.4.19;

contract DemoMemorytoMemoryReferenceTypeAssignment {

    uint stateVar = 20;
    function getUInt() returns (uint)
    {
        uint[] memory someVar = new uint[](1);

        someVar[0] = 23;

        uint[] memory  otherVar = someVar;

        someVar[0] = 45;

        return (otherVar[0]); //returns 45

    }
}
```

3.6　字面量

Solidity 为变量的赋值提供了字面量。字面量没有名字，它们本身就是值。变量可以在程序执行期间改变它们的值，但是字面量始终保持相同的值。看看下面的各种字面量的例子：

- 整数字面量的例子是 1、10、1 000、−1 和 −100。
- 字符串文字的例子是“Ritesh”和“Modi”。字符串字面量可以用单引号或双引号。
- 地址字面量的例子是 0xca35b7d915458ef540ade6068dfe2f44e8fa733c 和 0x1111111111111111111111111111111111111111。
- 十六进制字面量以前缀 hex 作为关键字。十六进制字面量的一个例子是 hex“1A2B3F”。

非卖品，仅供非商业用途或交流学习使用

- Solidity 支持使用点的十进制字面量。例子包括 4.5 和 0.2。

3.7 整型

整数有助于将数字存储在合约中。Solidity 提供以下两种类型的整数：

- **有符号的整数**：带符号的整数可以同时具有负值和正值。
- **无符号整数**：无符号整数只能保持正值和零。除正值和零值以外，它们也可以保持负值。

对于每种类型，Solidity 都有多种类型的整数。Solidity 提供了 uint8 类型来表示 8 位无符号整数，并且以 8 的倍数表示，直到达到 256。总之，可以声明 32 个不同的具有 8 的倍数的无符号整数，例如 uint8、uint16、unit24、uint256 位。同样，有符号整数的数据类型也是相同的，如 int8、int16，直到 int256。

根据要求，应选择适当大小的整数。例如，当存储 0 ～ 255 之间的值时，uint8 是合适的，而存储介于 −128 ～ 127 之间则 int8 更合适。对于更高的值，可以使用更大的整数。

有符号和无符号整数的缺省值为零，在声明时它们会自动初始化。整数是值类型；然而，当用作数组时，它们被称为参考类型。

可以对整数执行数学运算，例如加法、减法、乘法、除法、指数、否定、后增量和预增量。以下截图显示了其中一些示例：

```
pragma solidity 0.4.19;

contract AllAboutInts {

    uint  stateUInt = 20 ; //state variable
```

```
    uint  stateInt = 20 ; //state variable

    function getUInt(uint incomingValue)
    {
        uint memoryuint = 256 ;
        uint256 memoryuint256 = 256 ;
        uint8 memoryuint8 = 8 ; //can store value from 0 upto 255

        //addition of two uint8
        uint256 result = memoryuint8 + memoryuint ;

        // assignAfterIncrement = 9 and memoryuint8 = 9
       uint8 assignAfterIncrement = ++memoryuint8 ;

        // assignAfterIncrement = 9 and memoryuint8 = 10
      uint8 assignBeforeIncrement = memoryuint8++;


    }

    function getInt(int incomingValue)
    {
        int memoryInt = 256 ;
        int256 memoryInt256 = 256 ;
        int8 memoryInt8 = 8 ; //can store value from -128 to 127
    }

}
```

3.8　布尔型

像任何编程语言一样，Solidity 提供了一种布尔数据类型。`bool` 数据类型可用于表示具有二进制结果的场景，例如 `true` 或 `false`、1 或 0 等。此数据类型的有效值为 `true` 和 `false`。值得注意的是，Solidity 中的布尔不能转换为整数，就像它们在其他编程语言中一样。它是一个值类型，任何赋值给其他的布尔变量都会创建一个新副本。Solidity 中 `bool` 的默认值为 `false`。

声明和赋值 `bool` 数据类型的代码如下：

```
bool isPaid = true;
```

可以在合约中修改它，并可用于传入和传出参数以及返回值，如以下截图所示：

```
pragma solidity 0.4.19;

contract boolContract {

    bool isPaid = true;

    function manageBool() returns (bool)
    {
      isPaid = false;

      return isPaid; //returns false
    }

   function convertToUint() returns (uint8)
    {
      isPaid = false;

      return uint8(isPaid); //error
    }

}
```

3.9 字节数据类型

字节是指 8 位有符号整数。内存中的所有内容都存储在由二进制值 0 和 1 组成的位中。Solidity 还提供字节数据类型以存储二进制格式信息。通常，编程语言只有一种数据类型来表示字节。但是，Solidity 具有多种字节类型。它提供的数据类型范围为 `bytes1` ～ `bytes32`（含），以根据需要表示不同的字节长度。这些被称为**固定大小的字节数组**，并被实现为值类型。`bytes1` 数据类型代表 1 个字节，`bytes2` 代表 2 个字节。字节的默认值是 `0x00`，并用此值初始化。Solidity 也有一个 `byte` 类型，它是 `bytes1` 的别名。

一个字节可以以十六进制格式赋值，如下所示：

```
bytes1 aa = 0x65;
```

一个字节可以被赋值为十进制格式的整数值，如下所示：

```
bytes1 bb = 10;
```

一个字节可以被赋值为十进制格式的负整数值，如下所示：

```
bytes1 ee = -100;
```

一个字节可以赋值为字符值，如下所示：

```
bytes1 dd = 'a';
```

在下面的代码片段中，256 不适合放入单个字节，需要更大的字节数组：

```
bytes2 cc = 256;
```

下图中的代码显示了如何在固定大小的字节数组中存储二进制、正整数和负整数以及字符字面量。

我们还可以对字节数据类型执行按位操作，例如 `and`、`or`、`xor`、`not` 和左右移位操作：

```
pragma solidity 0.4.19;

contract bytesContract {

    bytes1 aa = 0x65;
    bytes1 bb = 10;
    bytes2 cc = 256;
    bytes1 dd = 'a';
    bytes1 ee = -100;

   function getintaa() returns (uint)
```

```
    {
      return uint(aa); //returns 101
    }

    function getbyteaa() returns (bytes1)
    {
      return aa; //returns 0x65
    }

    function getbytebb() returns (bytes1)
    {
      return bb; //returns 0x0a
    }

    function getintbb() returns (uint)
    {
      return uint(bb); //returns 10
    }

    function getbytecc() returns (bytes2)
    {
      return cc; //returns 0x0100
    }

    function getintcc() returns (uint)
    {
      return uint(cc); //returns 256
    }

    function getbytedd() returns (bytes2)
    {
      return dd; //returns 0x6100 or 0x61 for bytes1
    }

    function getintdd() returns (uint)
    {
      return uint(dd); //returns 97
    }
}
```

3.10 数组

数组是数据类型，但更具体地说，它们是依赖于其他数据类型的数据结构。

数组是指相同类型的数值组。数组有助于将这些值存储在一起，并简化迭代、排序和搜索该组中元素或子元素的过程。Solidity 提供了丰富的数组结构，可以满足不同的需求。

Solidity 中的数组示例如下：

```
uint[5] intArray ;
```

Solidity 中的数组可以是固定的或动态的。

3.10.1　固定数组

固定数组是指声明了预定大小的数组。固定数组的例子如下：

```
int[5] age ; // array of int with 5 fixed storage space allocated
byte[4] flags ; // array of byte with 4 fixed storage space allocated
```

固定数组无法使用 new 关键字进行初始化。它们只能以内联方式初始化，如下面的代码所示：

```
int[5] age = [int(10), 20,30,40,50] ;
```

它们也可以稍后在函数中内联初始化，如下所示：

```
int[5] age ;
age = [int(10),2,3,4,5];
```

3.10.2　动态数组

动态数组是指在声明时没有预定大小的数组，但是，它们的大小是在运行时确定的。看看下面的代码：

```
int[] age ; // array of int with no fixed storage space allocated. Storage
is allocated during assignment
byte[] flags ; // array of byte with no fixed storage space allocated
```

动态数组可以内联初始化或使用 new 运算符初始化。可以在声明时初始化，如下所示：

```
int[] age = [int(10), 20,30,40,50] ;
int[] age = new int[](5) ;
```

稍后，也可以在函数中以两个不同步骤初始化：

```
int[] age ;
age = new int[](5) ;
```

3.10.3 特殊数组

Solidity 提供了以下两个特殊数组：

- 字节数组
- 字符串数组

字节数组

字节数组是一个动态数组，可以容纳任意数量的字节。它与 `byte []` 不同。`byte []` 数组每个元素占用 32 个字节，而字节数组紧紧地将所有字节保存在一起。

字节可以声明为具有初始长度大小的状态变量，如以下代码所示：

```
bytes localBytes = new bytes(0) ;
```

这也可以分成与以前讨论的数组类似的以下两条代码行：

严禁网络传播本PDF，违者责任自负！


```
bytes localBytes ;
localBytes= new bytes (10) ;
```

字节数组可以直接赋值，如下所示：

```
localBytes = "Ritesh Modi";
```

此外，如果数据位于存储位置，则可以将值压栈其中，如下面的代码所示：

```
localBytes.push(byte(10));
```

字节数组还提供读 / 写长度属性，如下所示：

```
return localBytes.length; //reading the length property
```

请看下面的代码：

```
localBytes.length = 4; //setting bytes length to 4 bytes
```

字符串数组

字符串是基于上一节讨论的字节数组的动态数据类型。它们与附加约束的字节数组非常相似。字符串不能被索引或压栈，也不具有 `length` 属性。要对字符串变量执行任何这些操作，应首先将其转换为字节，然后在操作后将其转换回字符串。

字符串可以由单引号或双引号内的字符组成。字符串可以直接声明并赋值，如下所示：

```
String name = 'Ritesh Modi" ;
```

它们也可以转换为字节，如下所示：

严禁网络传播本PDF，违者责任自负！


```
Bytes byteName = bytes(name) ;
```

3.10.4 数组属性

数组支持一些基本的属性。在 Solidity 中，由于有多种类型的数组，并非每种类型都支持所有这些属性。

这些属性如下所示：

- index：除了字符串类型外，所有类型的数组都支持用于读取单个数组元素的 index 属性。仅动态数组，固定数组和字节类型支持用于写入单个数组元素的 index 属性。字符串和固定大小的字节数组不支持写入。
- push：仅动态数组支持此属性。
- length：除了字符串类型外，此属性由读取透视图中的所有数组支持。只有动态数组和字节支持修改长度属性。

3.11 数组的结构

我们已经简要介绍了结构的主题。结构有助于用户自定义数据结构。结构有助于将不同数据类型的多组变量转换为单一类型。结构不包含任何用于执行的编程逻辑或代码；它仅包含变量声明。结构是引用类型，在 Solidity 中被视为复杂类型。

结构可以定义为状态变量，如下代码所示。定义了一个由 string、uint、bool 和 uint 数组组成的结构体。有两个状态变量。在存储中保存它们。当第一个 `stateStructure1` 状态变量在声明时被初始化时，另一个 `stateStructure1` 状态变量留待稍后在函数中初始化。内存位置的局部结构在 `getAge` 函数中声明并初始化。声明另一个结构，作为指向 `stateStructure` 状态变量的指针。声明

严禁网络传播本PDF，违者责任自负！

了第三个本地结构，它指向之前创建的 localStructure 本地结构。在先前声明的状态结构被初始化的同时执行 localStructure 属性的改变，并最终返回来自 pointerLocalStructure 的年龄。它返回分配给 localStructure 的新值，如以下截图所示：

```
pragma solidity 0.4.19;

//contract definition
contract generalStructure {
    //state variables

    //structure definition
    struct myStruct {
        string name; //variable fo type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
    }

    // state structure
    myStruct  stateStructure = myStruct("Ritesh", 10, true, new uint[](2));

    myStruct  stateStructure1;


    //function definition
    function getAge ()  returns (uint) {

        // local structure
        myStruct memory localStructure = myStruct("Modi", 20 ,false, new uint[](2));

        //local pointer to State structure
        myStruct pointerStructure = stateStructure;

        // pointerlocalStructure is reference to localStructure
        myStruct memory pointerlocalStructure = localStructure;

        //changing value in localStructure
        localStructure.myAge = 30;

        //assigning values to state variable
        stateStructure1 =   myStruct("Ritesh", 10, true, new uint[](2));

        //returning pointerlocalStructure.Age -- returns 30
        return pointerlocalStructure.myAge;

    }
}
```

严禁网络传播本PDF，违者责任自负！

3.12 枚举

在本章前面讨论 Solidity 文件的布局时，我们简要地谈到了枚举的概念。枚举是包含一个预定义的常量值列表的值类型。它们通过值传递，每个副本都维护自己的值。不能在函数内声明枚举，并在合约的全局域命名空间内声明。

预定义的常量是连续赋值的，从零开始增加整数值。

下面显示的代码插图声明了一个枚举，该枚举被标识为由五个常量值组成的状态——`created`、`approved`、`provisioned`、`rejected` 和 `deleted`。将整数值 `0`、`1`、`2`、`3`、`4` 赋值给它们。

使用初始值 `provisioned`，创建了一个名为 `myStatus` 的枚举实例。

`returnEnum` 函数返回状态并返回整数值。值得注意的是，`web3` 和**去中心化应用**（DApp）不理解合约中声明的枚举。它们将得到一个对应于枚举常量的整数值。

`returnEnumInt` 函数返回一个整数值。

`passByValue` 函数展示了枚举实例维护自己的本地副本并且不与其他实例共享。

`assignInteger` 函数显示了一个例子，其中一个整数被赋值为一个枚举实例的值：

```
pragma solidity 0.4.19;

contract Enums {

//enum declared
```

严禁网络传播本PDF，违者责任自负！


```
enum status {created, approved, provisioned, rejected, deleted}

//instance of enum with initial value 2
status myStatus = status.provisioned;


    function returnEnum() returns (status)
    {
        status stat = status.created;
        return stat;
    }

   function returnEnumInt() returns (uint)
    {

       status stat = status.approved;
        return uint(stat);
    }

    function passByValue() returns (uint)
    {

       status stat = myStatus;
       myStatus = status.rejected;

        return uint(myStatus);
    }

    function assignInteger() returns (uint)
    {

       status stat = myStatus;

       //casting integer 2 to enum and assigning
       myStatus = status(2);

        return uint(myStatus);
    }
}
```

3.13　地址

地址是 20 字节的数据类型。它是为了存储以太坊中的账户地址而特别设计的，其大小为 160 位或 20 字节。它可以保存合约账户地址以及外部拥有的账户

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

地址。地址是一种值类型，它被赋值给另一个变量时会创建一个新副本。

地址具有 balance 属性，该属性返回账户可用的以太币数量，并具有一些用于账户间交易以太币和调用合约函数的功能。

它提供以下两个函数来交易以太币：

- transfer
- send

当向一个账户发送以太币时，更应该选择 transfer 函数而不是 send 函数。send 函数返回一个布尔值，具体取决于以太币发送是否成功执行，而 transfer 函数引发异常并将以太币返还给调用者。

它还提供了以下三个用于调用合约函数的函数：

- Call
- DelegateCall
- Callcode

3.14 映射

映射是 Solidity 中最常用的复杂数据类型之一。映射类似于其他语言中的散列表或字典。它们存储键值对，并允许根据提供的键来检索值。

使用 mapping 关键字声明映射，后跟由 => 表示法分隔的键和值的数据类型。映射具有与任何其他数据类型一样的标识符，并且它们可用于访问映射。

一个声明映射的例子如下：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
Mapping ( uint => address ) Names ;
```

在前面的代码中，uint 数据类型用于存储键而 address 数据类型用于存储值。Names 用作映射的标识符。

虽然它类似于散列表和字典，但 Solidity 不允许迭代映射。如果键已知，则可以检索映射中的值。下一个示例说明如何使用映射。合约中维护有一个 uint 类型的计数器作为映射的键，并且在函数的帮助下存储和检索地址详细信息。

要访问映射中的任何特定值，相关键应与映射名一起使用，如下所示：

```
Names[counter]
```

要在映射中存储值，请使用以下语法：

```
Names[counter] = <<some value>>
```

看看下面的截图：

```
pragma solidity 0.4.19;

contract GeneralMapping {

    mapping (uint => address) Names;

    uint counter;

    function addtoMapping(address addressDetails) returns (uint)
    {
        counter = counter + 1;
      Names[counter] = addressDetails;

      return counter; //returns false
    }

   function getMappingMember(uint id) returns (address)
    {
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
        return Names[id];
    }
}
```

虽然映射不支持迭代，但有一些方法可以解决这个限制。下一个示例说明了迭代映射的方法之一。请注意，在以太坊的 gas 使用方面，通常应该避免迭代和循环这类昂贵的操作。在此示例中，维护单独的计数器以跟踪映射中存储的条目的数量。此计数器还充当映射中的键。可以构造局部数组以存储来自映射的值。可以使用计数器执行循环，并可以将映射中的每个值提取并存储到本地数组中，如以下截图所示：

```
pragma solidity 0.4.19;

contract MappingLooping {

    mapping (uint => address) Names;

    uint counter;

    function addtoMapping(address addressDetails) returns (uint)
    {
      counter = counter + 1;

      Names[counter] = addressDetails;

      return counter;
    }

   function getMappingMember(uint id) returns (address[])
    {
      address[] memory localBytes = new address[](counter);
      for(uint i=1; i<= counter; i++){
          localBytes[i - 1] = Names[i];
      }

      return localBytes;
    }
}
```

只能将映射声明为 storage 类型的状态变量。不能在函数内将映射声明为内存映射。但是，如果映射引用状态变量中声明的映射，则可以在函数中声明映

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

射，如以下示例所示：

```
Mapping (uint => address) localNames = Names ;
```

这样的语法是合法的，因为 localNames 映射引用了 Names 状态变量：

```
pragma solidity 0.4.19;

contract MappinginMemory {

    mapping (uint => address) Names;

    uint counter;

    function addtoMapping(address addressDetails) returns (uint)
    {
      counter = counter + 1;
      mapping (uint => address) localNames = Names;

      localNames[counter] = addressDetails;

      return counter;
    }

   function getMappingMember(uint id) returns (address)
    {
      return Names[id];
    }
}
```

也可以使用嵌套映射，即由映射组成的映射。下一个例子说明了这一点。在此示例中，有一个显式的映射，将 uint 映射到另一个映射。子映射存储为第一个映射的值。子映射的键为 address 类型，值为 string 类型。有一个映射标识符，可以使用此标识符访问子映射或内部映射，如以下代码所示：

```
mapping (uint => mapping(address => string)) accountDetails;
```

要向此类嵌套映射添加条目，可以使用以下语法：

```
accountDetails[counter][addressDetails] = names;
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

这里，accountDetails 是映射标识符，counter 是父映射的键。accountDetails [counter] 映射标识符从父映射中检索值，返回的恰好是另一个映射。将键添加到返回值，我们可以设置内部映射的值。同样，可以使用以下语法检索内部映射的值：

```
accountDetails[counter][addressDetails]
```

截图如下所示：

```
pragma solidity 0.4.19;

contract DemoInnerMapping {

    mapping (uint => mapping(address => string)) accountDetails;
    uint counter;

    function addtoMapping(address addressDetails, bytes name) returns (uint)
    {
        string memory names = string(name);
        counter = counter + 1;
        accountDetails[counter][addressDetails] = names;

        return counter;
    }

   function getMappingMember(address addressDetails) returns (bytes)
    {
        // 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
      return bytes( accountDetails[counter][addressDetails]);
    }
}
```

3.15 本章小结

这是深入探索 Solidity 的第一章。本章介绍了 Solidity、Solidity 文件的布局，包括可以在其顶层声明的元素。从布局视角讨论了结构、预编译指令、合约和合同元素。本章的核心是完全沉浸在 Solidity 数据类型的世界里。深入讨

非卖品，仅供非商业用途或交流学习使用

论了值类型和引用类型以及 int、uint、固定大小的字节数组、字节、数组、字符串、结构、枚举、地址、布尔值和映射等类型，并结合示例进行了详细讨论。Solidity 为复杂类型，如结构和数组，提供了额外数据存储位置，深入讨论了这些管理其用法的规则。

在第 4 章中，我们将重点介绍智能合约的一些箱外变量和函数。Solidity 提供了许多全局变量和函数，以帮助简化获取当前交易和块上下文的任务。这些变量和函数提供上下文信息和 Solidity 代码，并将它们用于逻辑执行。它们在撰写企业级智能合约方面发挥着非常重要的作用。

非卖品，仅供非商业用途或交流学习使用

第 4 章 Chapter 4

全局变量和函数

在第 3 章中，你详细了解了 Solidity 的数据类型。数据类型可以是值或引用类型。有些引用类型（比如结构体和数组）也有与之关联的内存和存储。变量可以是状态变量或在函数内定义的本地变量。本章将重点介绍变量，其作用域规则、声明和初始化、类型转换和类型提升规则以及可对所有合约可见的全局变量。本章还将讨论一些全局函数。

4.1 var 类型变量

第 3 章中未讨论的一种 Solidity 类型是 var 数据类型。var 是一种只能在函数中声明的特殊类型。在合约中，状态变量不能是 var 类型。使用 var 类型声明的变量称为**隐式类型变量**，因为 var 不显式表示任何类型。它通知编译器它的类型是由第一次赋给它的值决定的。一旦确定类型后，则无法更改。

非卖品，仅供非商业用途或交流学习使用

编译器决定 var 变量的最终数据类型，而不是引用类型的开发人员。因此，由 block.difficulty（uint）当前区块编译器确定的类型可能不完全是代码执行所期望的类型。var 不能与显式内存位置一起使用。显式内存位置需要显式变量类型。

var 的示例如下屏幕截图所示。变量 uintVar8 的类型为 uint8，变量 uintVar16 的类型为 uint16，变量 intVar8 的类型为 int8（有符号整数），变量 intVar16 的类型为 int16（有符号整数），变量 boolVar 的类型为布尔，变量 stringVar 的类型为字符串，变量 bytesVar 的类型为字节，变量 arrayInteger 的类型为 uint8 数组，变量 arrayByte 的类型为 bytes10 数组：

```
pragma solidity 0.4.19;

contract VarExamples {

    function VarType()
    {
        var uintVar8 = 10; //uint8
        uintVar8 = 255; //256 is error

        var uintVar16 = 256; //uint16
        uintVar16 = 65535; //aaa = 65536; is error

       var intVar8 = -1; //int8 values -128 to 127

       var intVar16 = -129; //int16 values -32768 to 32767

        var boolVar = true;
        boolVar = false; // 10 is error, 0 is error, 1 is error, -1 is error


        var stringVar = "0x10"; // this is string memory
        stringVar = "10"; // cc =123123123123123123212222222 is error

        var bytesVar = 0x100; // this is byte memory


        var Var = hex"001122FF";

        var arrayInteger = [uint8(1),2];
        arrayInteger[1] = 255;
```

非卖品，仅供非商业用途或交流学习使用

```
        var arrayByte = bytes10(0x2222);
        arrayByte = 0x11111111111111111111; //0x11111111111111111111 is error
    }
}
```

4.2　变量声明提前

变量声明提前的概念是说在使用变量之前不需要声明和初始化变量。变量声明可以在函数内的任何地方发生，即使在使用它之后也可以。这就是所谓的**变量声明提前**。Solidity 编译器提取函数内任意位置声明的所有变量，并将它们放在函数的顶部或开头，我们都知道，在 Solidity 中声明变量会使用各自的默认值进行初始化。这确保了在整个函数中变量都可用。

在下面的示例中，`firstVar`、`secondVar` 和 `result` 在函数的末尾声明，但在函数的开头使用。然而，当编译器生成合约的字节码时，它将所有的变量声明作为函数中的第一组指令，如下面的屏幕截图所示：

```
pragma solidity ^0.4.19;

contract variableHoisting {

    function hoistingDemo() returns (uint){

        firstVar = 10;
        secondVar = 20;

        result = firstVar + secondVar;

        uint firstVar;
        uint secondVar;
        uint result;
```

非卖品，仅供非商业用途或交流学习使用

```
        return result;

    }
}
```

4.3 变量作用域

作用域是指在 Solidity 中合约和函数中的变量可用性。Solidity 提供以下两个可以声明变量的位置：

- 合约级全局变量——也称为状态变量
- 函数级局部变量

很容易理解函数级局部变量。它们仅在函数内可用，而在函数外部不可用。

合约级全局变量是可用于所有函数的变量，包括合约中的构造函数、fallback 函数和修改器。合约级全局变量可以附加一个可见性修改器。重要的是，无论可见性修改器如何，都可以在整个网络中查看状态数据。以下状态变量只能使用函数进行修改：

- `public`：可以直接从外部调用访问这些状态变量。编译器隐式生成 `getter` 函数以读取公共状态变量的值。
- `internal`：无法直接从外部调用访问这些状态变量。可以从当前合约中的函数和此合约的子合约访问它们。
- `private`：无法直接从外部调用访问这些状态变量，子合约也无法访问它们，只能通过当前合约中的函数访问。

看一下如下屏幕截图中的这些状态变量：

```
pragma solidity ^0.4.19;

contract ScopingDtateVariables {

//   uint64 public myVar = 0;

//   uint64 private myVar = 0;

//  uint64 internal myVar = 0;



}
```

4.4　类型转换

到目前为止，我们知道 Solidity 是一种静态类型语言，编译时会使用特定数据类型定义变量。在变量的生命周期内无法更改数据类型。这意味着只能存储合法的值。例如，uint8 可以存储 0 ～ 255 之间的值。它不能存储负值或大于 255 的值。为了更好地理解这一点，来看一下如下代码：

```
pragma solidity ^0.4.19;

contract ErrorDataType {

    function hoistingDemo() returns (uint){

        uint8 someVar = 100;
        someVar = 300; //error


    }

}
```

但是，有时复制一种类型的变量到另一种类型的变量需要做某种转换，这称为**类型转换**。Solidity 提供了类型转换的规则。

在 Solidity，可以执行各种转换，我们将在以下部分介绍这些内容。

4.4.1 隐式转换

隐式转换意味着不需要操作符，或者不需要外部帮助进行转换。这些类型的转换是完全合法的，数据不会丢失或者值不匹配。它们完全是类型安全的。Solidity 允许从较小的整数类型到较大的整数类型的隐式转换。例如，将 uint8 转换为 uint16 会隐式发生。

4.4.2 显式转换

当由于数据丢失或包含不在目标数据类型范围的数据的值时，编译器不执行隐式转换，此时需要显式转换。Solidity 为每种值类型提供了显式转换的函数。显式转换的示例是 `uint16` 转换为 `uint8`。在这种情况下可能会丢失数据。

以下代码清单显示了隐式和显式转换的示例：

- `ConvertionExplicitUINT8toUINT256`：此函数执行从 `uint8` 到 `uint256` 的显式转换。需要注意的是，这种转换也可能是隐式的。
- `ConvertionExplicitUINT256toUINT8`：此函数执行从 `uint256` 到 `uint8` 的显式转换。如果转换是隐式发生的，则此转换将引发编译时错误。
- `ConvertionExplicitUINT256toUINT81`：此函数展示了显式转换有趣的一面。显式转换容易出错，通常应该避免。在此函数中，尝试将较大的值存储在较小数据类型的变量中。这可能导致数据丢失和不可预

测性。编译器不会生成错误，但是，它会尝试将值拟合为较小的值并循环查找有效值。

- `Conversions`：此函数展示了隐式和显式转换的示例。代码中有些是失败代码，有些是合法的。在如下的屏幕截图中，请阅读代码下方的注释以理解它们。

```
pragma solidity ^0.4.19;

contract ConversionDemo {

    function ConvertionExplicitUINT8toUINT256() returns (uint){
        uint8 myVariable = 10;
        uint256 someVariable = myVariable;
        return someVariable;
    }

    function ConvertionExplicitUINT256toUINT8() returns (uint8){
        uint256 myVariable = 10;
        uint8 someVariable = uint8(myVariable);
        return someVariable;
    }

    function ConvertionExplicitUINT256toUINT81() returns (uint8){
        uint256 myVariable = 10000134;
        uint8 someVariable = uint8(myVariable);
        return someVariable; //returns 6 as return value
    }

    function Convertions() {

        uint256 myVariable = 10000134;
        uint8 someVariable  = 100;
        bytes4 byte4 = 0x65666768;

        // bytes1 byte1 = 0x656667668; //error

        bytes1 byte1 = 0x65;

        //  byte1 = byte4; //error, explicit conversion needed here

        byte1 = bytes1(byte4) ; //explicit conversion

        byte4 = byte1;  //Implicit conversion

        // uint8 someVariable = myVariable; // error, explicit conversion needed here

        myVariable = someVariable; //Implicit conversion

        string memory name = "Ritesh";
        bytes memory nameInBytes = bytes(name); //explicit string to bytes conversion

        name = string(nameInBytes); //explicit bytes to string conversion

    }
}
```

4.5 区块和交易全局变量

Solidity 提供了几个全局变量的访问权限，这些变量未在合约中声明，但可以通过合约代码访问。合约无法直接访问账本。账本仅由矿工维护。但是 Solidity 提供了一些当前交易和区块的相关信息，以便可以使用它们。Solidity 提供了区块以及与交易相关的变量。

以下代码示例介绍了如何使用全局交易、区块和消息变量：

```
pragma solidity ^0.4.19;

contract TransactionAndMessageVariables {

    event logstring(string);
    event loguint(uint);
    event logbytes(bytes);
    event logaddress(address);
    event logbyte4(bytes4);
    event logblock(bytes32);

    function globalVariable() payable {

        logaddress( block.coinbase ); // 0x94d76e24f818426ae84aa404140e8d5f60e10e7e

        loguint( block.difficulty ); //71762765929000

        loguint( block.gaslimit ); // 6000000

        loguint(  msg.gas ); //2975428

        loguint(  tx.gasprice ); // 1

        loguint(  block.number ); //123

        loguint( block.timestamp ); //1513061946

        loguint( now ); //1513061946

        logbytes( msg.data ); // 0x4048d797

        logbyte4(   msg.sig  ); // // 0x4048d797

        loguint( msg.value ); // 0 or in Wei if ether are send

        logaddress( msg.sender ); //0xca35b7d915458ef540ade6068dfe2f44e8fa733c"

        logaddress( tx.origin ); // 0xca35b7d915458ef540ade6068dfe2f44e8fa733c"

        logblock ( block.blockhash( block.number) ); //0x0000000000000000000000

    }
}
```

4.5.1　交易和消息全局变量

以下是全局变量及其数据类型的列表以及提供的参考描述：

变量名称	描述
block.coinbase (address)	与 etherbase 相同，指矿工的地址
block.difficulty (uint)	当前区块难度等级
block.gaslimit (uint)	当前区块 gas 限制
block.number (uint)	顺序表示的区块编号
block.timestamp (uint)	区块创建时间
msg.data (bytes)	与创建交易相关的函数及其参数信息
msg.gas (uint)	执行交易后未花费 gas
msg.sender (address)	调用函数的地址
msg.sig (bytes4)	函数标识符使用散列函数签名后的前四个字节
msg.value (uint)	随交易发送的以太币数量，以 wei 作为单位
now (uint)	当前时间
tx.gasprice (uint)	调用者准备支付的每一种 gas 单位下的 gas 价格
tx.origin (address)	交易的发起者
block.blockhash(uint blockNumber) returns (bytes32)	包含交易的区块散列值

4.5.2　tx.origin 和 msg.sender 的区别

细心的读者可能已经注意到在之前的代码例子中，tx.origin 和 msg.sender 的结果和输出是相同的。tx.origin 全局变量是指发起交易的原始外部账户，而 msg.sender 是指调用该函数的直接账户（可以是外部账户或其他合约账户）。tx.origin 变量将始终引用外部账户，而 msg.sender 可以是合约内账户或外部账户。如果多个合约上有多个函数调用，则 tx.origin 将始终引用发起交易的账户，而不管调用的合约堆栈如何。但是，msg.sender 将引用调用下一个合约的先前一个账户（合约内 / 外部）。建议优先使用 msg.sender。

4.6 加密全局变量

Solidity 为合约提供了散列值的加密函数。有两个散列函数—— SHA2 和 SHA3。将输入转换为散列有两个函数，分别是基于 sha3 算法的 sha3 函数和基于 sha2 算法的 sha256 函数。还有另一个函数 keccak256，它是 SHA3 算法的别名。建议当有需要使用散列的时候调用 keccak256 或 sha3 函数。

下面的代码段截图说明了这一点：

```
pragma solidity ^0.4.19;

contract CryptoFunctions {

    function cryptoDemo() returns (bytes32, bytes32, bytes32){

        return (sha256("r"), keccak256("r"), sha3("r"));


    }
}
```

此函数执行的结果如下面的截图所示。keccak256 和 sha3 函数的执行结果是一样的。

```
{
        "0": "bytes32: 0x454349e422f05297191ead13e21d3db520e5abef52055e4964b82fb213f593a1",
        "1": "bytes32: 0x414f72a4d550cad29f17d9d99a4af64b3776ec5538cd440cef0f03fef2e9e010",
        "2": "bytes32: 0x414f72a4d550cad29f17d9d99a4af64b3776ec5538cd440cef0f03fef2e9e010"
}
```

所有这三个函数都接收紧密打包的参数，这意味着可以将多个参数连接在一起以查找散列，如下面的代码片段所示：

```
ConvertionExplicitUINT8toUINT256:
```

严禁网络传播本PDF，违者责任自负！

4.7　地址全局变量

外部的或基于合约的每个地址，都具有五个全局函数和一个全局变量。在随后的 Solidify 函数章节中我们将对这些函数和变量深入讨论。与地址相关的全局变量称为**余额**，它提供地址中以 wei 为单位的以太币余额。

函数如下：

- `<address>.transfer(uint256 amount)`：该函数向 `address` 发送给定的以 wei 为单位的以太币，如果执行失败则返回 `false`。
- `<address>.send(uint256 amount)`：该函数向 `address` 发送给定的以 wei 为单位的以太币，并在失败时返回 `false`。
- `<address>.call(...) returns (bool)`：此函数调用低级别的 `call` 函数，并在失败时返回 `false`。
- `<address>.callcode(...) returns (bool)`：此函数调用低级别的 `callcode` 函数，失败时返回 `false`。
- `<address>.delegatecall(...) returns (bool)`：此函数调用低级别的 `delegatecall` 函数，失败时返回 `false`。

4.8　合约全局变量

每份合约都有以下三个全局函数：

- `this`：当前合约的类型，可显式转换为地址。
- `selfdestruct`：接收一个地址，销毁当前合约，将其资金发送到给定地址。
- `suicide`：接收一个地址，也是 `selfdestruct` 的别名。

严禁网络传播本PDF，违者责任自负！

4.9 本章小结

本章在许多方面是前几章的延续。前半部分对变量进行了深入讨论，详细阐述了变量声明提前、类型转换、var 数据类型和 Solidity 变量的范围，以及代码示例。本章的后半部分侧重于全局变量和函数。对交易和消息相关的变量，例如 block.coinbase，msg.data 等，进行了解释。本章还介绍了 msg.sender 和 tx.origin 之间的区别及其用法。还讨论了加密、地址和合约级函数。我们将在另一章中重点介绍这些函数。

第 5 章将重点介绍 Solidity 表达式和控制结构，包括有关循环和条件的编程细节。这将是一个重要章节，因为每个程序都需要某种循环来执行重复任务，而 Solidity 控制结构有助于实现这些。循环基于条件，条件使用表达式编写。这些表达式被计算并返回 true 或 false。请继续关注我们在第 5 章中介绍的控制结构和表达式。

严禁网络传播本PDF，违者责任自负！

第 5 章 Chapter 3

表达式和控制结构

编程语言的一个重要方面是在代码中做出决策，Solidity 也应该能够根据环境执行不同的指令。Solidity 为此提供 `if...else` 和 `switch` 语句。循环遍历多个元素也很重要，Solidity 为此提供了多个控制结构，例如 `for` 循环和 `while` 语句。在本章中，我们将详细讨论编程结构，以帮助你做出决策和循环遍历一组值。

5.1 Solidity 表达式

表达式是指一个语句（包含多个操作数和可选的零个或多个运算符），它们生成单个值，对象或函数。操作数可以是字面量、变量、函数调用或其他表达式。

表达式的示例如下：

严禁网络传播本PDF，违者责任自负！


```
Age>10
```

在上面的代码中，Age 是一个变量，10 是一个整数字面量。Age 和 10 是操作数，(>) 大于符号是操作符。此表达式返回单个布尔值（true 或 false），具体取决于 Age 中存储的值。

表达式可能更复杂，包括多个操作数和运算符，如下所示：

```
((Age > 10) && (Age < 20) ) || ((Age > 40) && (Age < 50) )
```

在前面的代码中，有多个运算符在起作用。&& 运算符充当两个表达式之间的 AND 运算符，这两个表达式又包含操作数和运算符。两个复杂表达式之间还有一个由 || 表示的 OR 运算符。

Solidity 具有以下比较运算符，可帮助编写返回布尔值的表达式：

操作符	含义	示例
==	相等	myVar==10
!=	不相等	myVar!=10
>	大于	myVar>10
<	小于	myVar<10
>=	大于或者等于	myVar>=10
<=	小于或者等于	myVar<=10

Solidity 还提供以下逻辑运算符，帮助编写返回布尔值的表达式：

运算符	含义	示例
&&	与	(myVar>10)&&(myVar<10)
||	或	myVar!=10
!	非	myVar>10

与其他语言一样，以下运算符在 Solidity 中具有优先级：

严禁网络传播本PDF，违者责任自负！

优先级	描述	运算符
1	后缀自增和自减	++、--
新建表达式	new<typename>	NA
数组下标	<array>[<index>]	NA
成员访问	<object>.<member>	NA
函数调用	<func>(<args...>)	NA
圆括号	(<statement>)	NA
2	前缀自增和自减	++、--
一元正负号	+、-	NA
一元运算	delete	NA
逻辑非	!	NA
按位取反	~	NA
3	求幂	**
4	乘法、除法和模运算	*、/、%
5	加法、减法	+、-
6	按位移位	<<、>>
7	按位与	&
8	按位异或	^
9	按位或	|
10	不等式操作符	<、>、<=、>=
11	等式操作符	==、!=
12	逻辑或	&&
13	逻辑与	||
14	三元算子	<conditional>?<if-true>:<if-false>
15	赋值操作符	=、|=、^=、&=、<<=、>>=、+=、-=、*=、/=、%=
16	逗号操作符	,

5.2　if 决策控制

Solidity 在 if...else 指令的帮助下提供条件代码执行。if...else 的一般结构如下：

严禁网络传播本PDF，违者责任自负！


```
if (this condition/expression is true) {
   Execute the instructions here
}
else if (this condition/expression is true) {
   Execute the instructions here
}
else {
   Execute the instructions here
}
```

if 和 if-else 是 Solidity 中的关键字，它们通知编译器其中包含决策控制条件，例如 if（a>10）。这里，if 包含可以计算为 true 或 false 的条件。如果 a>10 的计算结果为 true，则应执行括号（{}）中的代码指令。

else 也是一个关键字，如果以前的条件均不为真，则提供备用路径。它还包含一个决策控制指令，并在 >10 成立时执行代码指令。

以下示例显示了“IF”-“ELSEIF”-“ELSE”条件的用法。声明了具有多个常量的枚举。StateManagement 函数接受一个 uint8 参数，该参数被转换为枚举常量并在 if...else 决策控制结构中进行比较。如果值为 1 则返回结果为 1；如果参数包含 2 或 3 作为值，则 else...if 部分代码被执行；如果值不是 1、2 或 3，则执行 else 部分：

```
pragma solidity ^0.4.19;

contract IfElseExample {

    enum requestState {created, approved, provisioned, rejected, deleted, none}

    function StateManagement(uint8 _state) returns (int result) {

       requestState currentState = requestState(_state);

       if(currentState == requestState(1)){
           result = 1;
       } else if ((currentState == requestState.approved) || (currentState == requestState.provisioned)) {
           result = 2;
       } else {
           currentState == requestState.none;
           result = 3;
       }

    }
}
```

严禁网络传播本PDF，违者责任自负！

5.3　while 循环

有时我们需要根据条件重复执行代码段。为此目的，Solidity 提供了精确的 while 循环。while 循环的一般形式如下：

```
Declare and initialize a counter
while (check the value of counter using an expression or condition) {
    Execute the instructions here
    Increment the value of counter
}
```

while 是 Solidity 中的关键字，它通知编译器其包含一个决策控制指令。如果此表达式的计算结果为 true，那么应该执行括号（{}）中的代码指令。while 循环保持执行，直到条件变为 false。

在以下示例中，声明了 mapping 和 counter。counter 用于循环 mapping，因为在 Solidity 中循环 mapping 没有开箱即用的支持。

事件用于获取有关交易的详细信息。我们将在第 8 章中详细讨论。现在，只要知道你在调用事件时记录信息就足够了。SetNumber 函数将数据添加到映射，getnumbers 函数运行 while 循环以检索映射中的所有条目并使用事件记录它们。

注意：临时变量用作计数器，在每次执行 while 循环时加 1。

while 条件检查临时变量的值，并将其与全局计数器变量进行比较。根据它是真还是假，执行 while 循环中的代码。在这组指令中，应修改计数器的值，以便通过使 while 条件变为 false 来帮助退出循环，如以下屏幕截图所示：

```
pragma solidity ^0.4.19;

contract whileLoop {
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);
    bytes aa;

    function SetNumber()  {
        blockNumber[counter++] = block.number;
    }

    function getNumbers() {
       uint i = 0;
       while (i < counter) {
           uintNumber( blockNumber[i]  );
           i = i + 1;
       }

    }
}
```

5.4 for 循环

最著名和最常用的循环之一是 for 循环，我们可以在 Solidity 中使用它。for 循环的一般结构如下：

```
for (initialize loop counter; check and test the counter; increase the
value of counter;) {
     Execute multiple instructions here
  }
```

for 是 Solidity 中的关键字，它通知编译器其包含有关循环一组指令的信息。它与 while 循环非常相似。但它更简洁，更易读，因为所有信息都可以在一行中查看。

以下代码示例展示了相同的解决方案：循环遍历映射。但是，它使用 for 循环代替 while 循环。初始化 i 变量，在每个迭代器中递增 1，并检查它是否小于 counter 的值。一旦条件变为 false，循环就会停止。也就是说，i 的值

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

等于或大于 counter。

```
pragma solidity ^0.4.19;


contract ForLoopExample {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function SetNumber()  {

        blockNumber[counter++] = block.number;

    }

    function getNumbers() {

        for (uint i=0; i < counter; i++){
            uintNumber( blockNumber[i]  );
        }

    }
}
```

5.5　do...while 循环

do...while 循环与 while 循环非常相似，其一般形式如下：

```
Declare and Initialize a counter
do {
Execute the instructions here
Increment the value of counter
} while(check the value of counter using an expression or condition)
```

如果你注意到了，while 和 do...while 循环之间存在细微差别。do...while 中的条件位于循环指令的末尾。如果条件为假，则根本不执行 while 循环中的指令。但是，在计算条件之前，do...while 循环中的指令会执行一

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

次。因此，如果你想至少执行一次指令，那么与 while 循环相比，应该首选 do...while 循环。看一下代码片段的屏幕截图：

```
pragma solidity ^0.4.19;


contract DowhileLoop {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);
    bytes aa;

    function SetNumber()  {

        blockNumber[counter++] = block.number;


    }

    function getNumbers() {

       uint i = 0;
       do  {
           uintNumber( blockNumber[i]  );
           i = i + 1;
       } while (i < counter);

    }
}
```

5.6 break 语句

循环有助于在向量数据类型上从开始进行迭代。但是，有时你希望停止迭代，跳出或退出循环而不再执行条件测试。break 语句帮助我们做到这一点。它通过将控件传递给循环后的第一条指令来帮助我们终止循环。

在下面的截图示例中，由于使用了 break 语句，当 i 的值为 1 时，for 循环终止并且控制移出 for 循环。它实际上打破了循环，如下面的截图所示：

非卖品，仅供非商业用途或交流学习使用

```
pragma solidity ^0.4.19;


contract ForLoopExampleBreak {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function SetNumber()  {

        blockNumber[counter++] = block.number;


    }

    function getNumbers() {

        for (uint i=0; i < counter; i++){
            if (i == 1)
                break;
            uintNumber( blockNumber[i]  );

        }

    }
}
```

5.7　continue 语句

循环是基于表达式的。表达式的逻辑决定了循环的连续性。但是，有时你处于循环执行之间并希望返回到第一行代码而不执行下一次迭代的其余代码。`continue` 语句帮助我们做到这一点。

在下面的截图中，`for` 循环执行到结束，但是 5 之后的值根本没有记录：

```
pragma solidity ^0.4.19;

contract ForLoopExampleContinue {

    mapping (uint => uint) blockNumber;
    uint counter;
```

非卖品，仅供非商业用途或交流学习使用

```
    event uintNumber(uint);

    function SetNumber()  {

        blockNumber[counter++] = block.number;


    }

    function getNumbers() {

        for (uint i=0; i < counter; i++){
            if ((i > 5) )
              { continue;}
            uintNumber( blockNumber[i]  );

        }

    }
}
```

5.8 return 语句

返回数据是 Solidity 函数的组成部分。Solidity 为函数返回数据提供了两种不同的语法。在下面的代码示例中，定义了两个函数——`getBlockNumber` 和 `getBlockNumber1`。`getBlockNumber` 函数返回一个 `uint` 而不命名 `return` 变量。在这种情况下，开发人员可以显式地使用 `return` 关键字从函数返回。

`getBlockNumber1` 函数返回 `uint`，并提供变量的名称。在这种情况下，开发人员可以直接使用并从函数返回此变量，而不使用 `return` 关键字，如以下屏幕截图所示：

```
pragma solidity ^0.4.19;

contract ReturnValues {
```

非卖品，仅供非商业用途或交流学习使用

```
    uint counter;

    function SetNumber()  {

       counter = block.number;


    }

    function getBlockNumber() returns (uint) {

        return counter;

    }

    function getBlockNumber1() returns (uint result) {

        result =  counter;

    }
}
```

5.9　本章小结

表达式和控制结构是任何编程语言的组成部分，它们也是 Solidity 语言的重要元素。Solidity 为决策和循环结构提供了丰富的基础结构。它提供了 `if...else` 决策控制结构和 `for`、`do...while` 和 `while` 循环，用于循环遍历可迭代的数据变量。Solidity 还允许我们编写条件和逻辑、赋值以及编程语言支持的任何类型的语句。

后续章节将详细讨论 Solidity 和合约函数，这些是编写合约的核心要素。区块链是关于执行和存储交易，并且在执行合约函数时创建交易。函数可以更改以太坊的状态或只返回当前状态。更改状态的函数和返回当前状态的函数将在第 6 章中详细讨论。

非卖品，仅供非商业用途或交流学习使用

第 6 章 Chapter 6

编写智能合约

使用 Solidity 编写智能合约。本章专注于智能合约。从这章开始，你将开始编写智能合约。本章将从设计方面讨论编写智能合约，定义和实现合约以及使用不同机制部署和创建合约——使用新关键字和已知地址。Solidity 提供了丰富的面向对象设计，本章将深入研究面向对象的概念和实现，例如继承、多继承、声明抽象类和接口、以及为抽象函数和接口提供方法实现。

6.1 智能合约

什么是智能合约？每个人都试图理解合约的含义以及“智能”一词在合约中的重要性。智能合约本质上是在 EVM 中部署和执行的代码段或程序。合约是法律界通常使用的术语，与编程世界没有多大关系。在 Solidity 中写一份智能合约并不意味着写一份法律合约。此外，合约就像任何其他编程代码一样，包含 Solidity 代码，并在有人调用它们时执行。这本质上没什么智能。智能合

非卖品，仅供非商业用途或交流学习使用

约是区块链术语；它是一句行话，用于指代在 EVM 中执行的编程逻辑和代码。

智能合约与 C++、Java 或 C # 中的类很相似。就像一个类由状态（变量）和行为（方法）组成一样，合约包含状态变量和函数。状态变量的目的是维护合约的当前状态，并且函数负责执行逻辑并对当前状态执行更新和读取操作。

在第 5 章我们已经看到了一些智能合约的例子，然而，是时候深入研究这个主题了。

6.2 编写一个简单的合约

使用 `contract` 关键字和标识符来声明合约，如以下代码所示：

```
contract SampleContract {
}
```

括号内是状态变量和函数定义的声明。在第 3 章中讨论了合约的完整定义，我再次提供它以便快速参考。此合约包含状态变量、结构定义、枚举声明、函数定义、修改器和事件。第 4 章中详细讨论了状态变量、结构和枚举。函数、修改器和事件将在接下来的两章中详细讨论。看一下描述合约的代码段的以下截图：

```
pragma solidity 0.4.19;

//contract definition
contract generalStructure {
    //state variables
    int public stateIntVariable; // vriable of integer type
    string stateStringVariable; //variable of string type
    address personIdentifier; // variable of address type
    myStruct human; // variable of structure type
    bool constant hasIncome = true; //variable of constant nature

    //structure definition
    struct myStruct {
        string name; //variable fo type string
```

```
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
    }

    //modifier declaration
    modifier onlyBy(){
        if (msg.sender == personIdentifier) {
            _;
        }
    }

    // event declaration
    event ageRead(address, int );

    //enumeration declaration
    enum gender {male, female}

    //function definition
    function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

        human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

        gender _gender = gender.male; //using enum

        ageRead(personIdentifier, stateIntVariable);
    }

}
```

6.3　创建合约

在 Solidity 中有两种方法创建并使用合约：

- 使用 new 关键字
- 使用已经部署的合约地址

6.3.1　使用 new 关键字

在 Solidity 中使用 new 关键字部署并创建新的合约实例。通过部署合约来初始化合约实例，初始化状态变量，运行其构造函数，将 nonce 值设置为 1，并最终将实例的地址返回给调用者。部署合约涉及检查请求者是否提供了足够的 gas 来完成部署，使用请求者的 address 和 nonce 值生成合约部署的新账户 / 地址，并花费该地址上的以太币。

在下一个屏幕截图中，定义了两个合约，HelloWorld 和 client。在这种情况下，一个合约（client）部署并创建另一个合约（HelloWorld）的新实例。它使用 new 关键字执行此操作，如以下代码段所示：

```
HelloWorld myObj = new HelloWorld();
```

我们来看看下面的截图：

```
pragma solidity 0.4.19;

contract HelloWorld {
    uint private simpleInt;

    function   getValue() public view returns (uint) {
        return simpleInt;
    }

    function  setValue(uint _value) public {
        simpleInt =  _value;
    }
}


contract client  {

    function useNewKeyword() public  returns (uint) {

       HelloWorld myObj = new HelloWorld();

       myObj.setValue(10);

       return myObj.getValue();

    }
}
```

6.3.2 使用合约地址

当已经部署和实例化合约时，使用此方法来创建合约实例。这种方法使用现有的已部署合约的地址。没有创建新实例，相反，现有实例被重用。使用其地址对现有合约进行引用。

在下一个代码示例中，定义了两个合约 HelloWorld 和 client。在这

种情况下，一个合约（client）使用另一个合约的已知地址来创建对它的引用（HelloWorld）。它使用地址数据类型并将实际地址转换为 HelloWorld 合约类型。myObj 对象包含现有合约的地址，如以下代码段所示：

```
HelloWorld myObj = HelloWorld(obj);
```

我们来看看下面的截图：

```
pragma solidity 0.4.19;

contract HelloWorld {

    uint private simpleInt;

    function   GetValue() public view returns (uint) {
        return simpleInt;
    }

    function  SetValue(uint _value) public {
        simpleInt =  _value;
    }
}


contract client  {

    address obj ;

    function setObject(address _obj) external   {
        obj = _obj;
    }

    function UseExistingAddress() public  returns (uint) {
       HelloWorld myObj = HelloWorld(obj);
       myObj.SetValue(10);
       return myObj.GetValue();
    }
}
```

6.4　构造函数

Solidity 支持在合约中声明构造函数。在 Solidity 中构造函数是可选的，当没有显式定义构造函数时，编译器会合成默认构造函数。构造函数在部署合约

时执行一次。这与其他编程语言截然不同。在其他编程语言中，只要创建新的对象实例，就会执行构造函数。但是，在 Solidity 中，执行的构造函数部署在 EVM 上。构造函数应该初始化状态变量，通常应该避免编写大量的 Solidity 代码。构造函数代码是合约执行的第一组代码。与其他编程语言中的构造函数不同，合约中最多只能有一个构造函数。可以在部署合约时为构造函数提供参数。

构造函数与合约同名。它们的名字应该是一样的。从可见性的角度来看，构造函数可以是 `public` 也可以是 `internal`。但不能是 `external` 或 `private`。构造函数不会显式返回任何数据。在以下示例中，定义了与 `HelloWorld` 合约名称相同的构造函数。存储变量值设置为 5，如以下屏幕截图所示：

```
pragma solidity 0.4.19;

contract HelloWorld {

    uint private simpleInt;

    function HelloWorld() public {
        simpleInt = 5;
    }

    function   GetValue() public view returns (uint) {
        return simpleInt;
    }

    function  SetValue(uint _value) public {
        simpleInt =  _value;
    }
}
```

6.5 合约组合

Solidity 支持合约组合。组合是指将多个合约或数据类型组合在一起以创建复杂的数据结构和合约。我们之前已经看过许多合约组合的例子。请参阅使用

本章前面所示的 new 关键字创建合约的代码段。在此示例中，client 合约由 HelloWorld 合约组成。在这里，HelloWorld 是一个独立的合约，client 是一个依赖合约。因为它的完整性依赖于 HelloWorld 合约。将问题分解为多合约解决方案并使用合约组合是一种很好的做法。

6.6　继承

继承是面向对象的支柱之一，而 Solidity 支持智能合约之间的继承。继承是定义彼此相关的多个合约的父子关系的过程。被继承的合约称为**父合约**，继承的合约称为**子合约**。同样，有父合约的合约称为**派生合约**，父合约称为**基合约**。继承主要是关于代码的可重用性。基合约和派生合约之间存在 is-a 关系，所有公共和内部范围函数和状态变量都可用于派生合约。实际上，Solidity 编译器将基合约字节码复制到派生合约字节码中。is 关键字用于在派生合约中继承基合约。

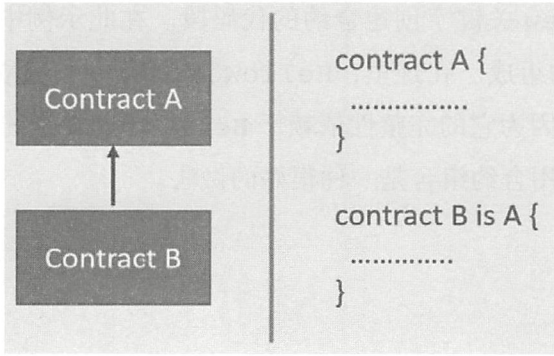
由于合约的版本化和部署方式，这是每个 Solidity 开发人员应该掌握的最重要的概念之一。

Solidity 支持多种类型的继承，包括多重继承。

Solidity 将基合约复制到派生合约中，并使用继承创建单个合约。生成单个地址，该地址在父子关系的合约之间共享。

6.6.1　单继承

单继承有助于将基合约的变量、函数、修改器和事件继承到派生合约中。看一下下图：



下一个代码片段有助于解释单继承。你会发现有两个合约，ParentContract 和 ChildContract。ChildContract 合约继承自 ParentContract。ChildContract 将继承所有公共和内部变量和函数。任何使用 ChildContract 的人，如在 client 合约中所见，都可以调用 GetInteger 和 SetInteger 函数，就像它们是在 ChildContract 中定义一样，如下面的屏幕截图所示：

```
pragma solidity 0.4.19;

contract ParentContract {
    uint internal simpleInteger;

    function SetInteger(uint _value) external {
        simpleInteger =  _value;
    }
}


contract ChildContract is ParentContract  {
    bool private simpleBool;

    function   GetInteger() public view returns (uint) {
        return simpleInteger;
    }

}



contract Client  {
```

```
    ChildContract pc = new ChildContract();

    function workWithInheritance() public  returns (uint) {
        pc.SetInteger(100);
        return pc.GetInteger();
    }

}
```

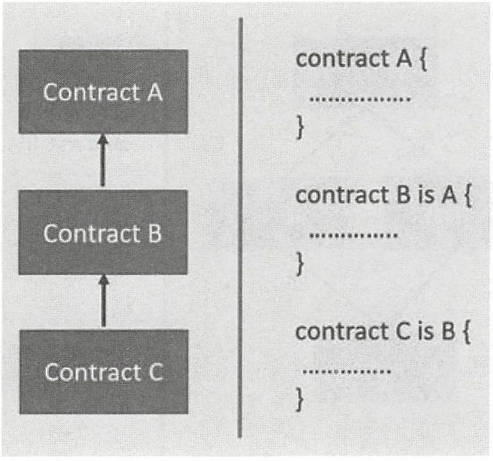
Solidity 合约中的所有函数都是虚函数，并且基于合约实例。在基类或派生类中调用合适的函数。此主题称为**多态**，本章后面的部分将对此进行介绍。

合约构造函数的调用顺序是从继承层次的最上层合约到最远派生合约。

6.6.2　多级继承

多级继承与单继承非常相似。但是，不仅仅是一个父子关系，还有多个级别的父子关系。

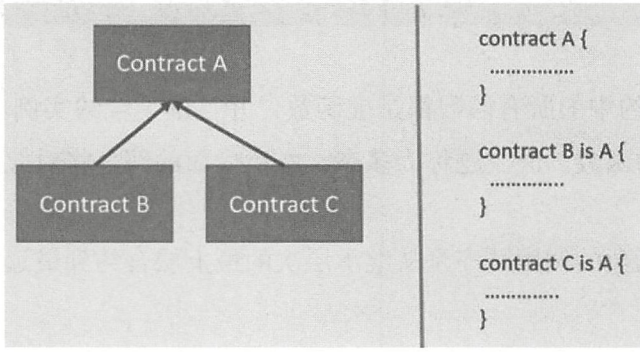
如下图所示。**合约 A** 是**合约 B** 的父合约，而**合约 B** 是**合约 C** 的父合约。



严禁网络传播本PDF，违者责任自负！

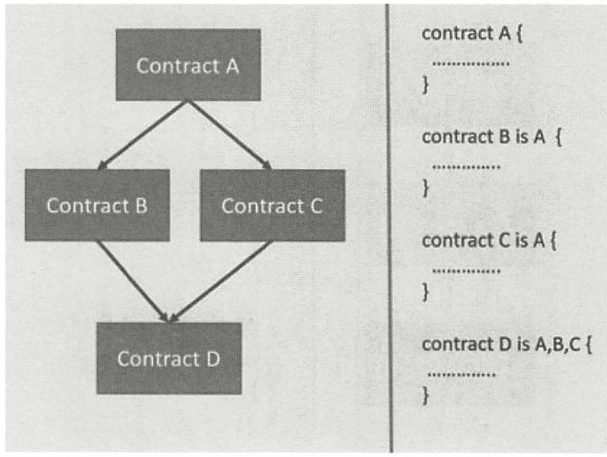
6.6.3 分层继承

分层继承也类似于单继承。但是，在这里，单个合约充当多个派生合约的基合约。如下图所示。**合约 A 被合约 B 和合约 C 同时继承：**



6.6.4 多重继承

Solidity 支持多重继承。单个继承可以有多个级别。但是，也可能存在多个相同基合约的派生合约。这些派生合约可以一起作为深一层子类的基合约。当从这些子合约一起继承时，发生多重继承，如下图所示：



严禁网络传播本PDF，违者责任自负！

下一个屏幕截图显示了多重继承的示例。在此示例中，SumContract 作为 MultiContract 和 DivideContract 合约的基合约。SumContract 合约提供了 Sum 函数的实现，MultiContract 和 DivideContract 合约分别提供了 Multiply 和 Divide 函数的实现。SubContract 继承自 MultiContract 和 DivideContract。SubContract 合约提供 Sub 函数的实现。client 合约不是父子层次结构的一部分，并且正在消费其他合约。client 合约创建 SubContract 的实例并在其上调用 Sum 方法。

Solidity 遵循 Python 的路径并使用 **C3 线性化**，也称为**方法解析顺序**（MRO），来强制基合约图中的特定顺序。合约应该在继承时遵循特定的顺序，从基合约到最远派生的合约。接下来显示了这种顺序的一个例子，其中 SubContract 合约继承自 SumContract、DivideContract 和 MultiContract。

以下屏幕截图的代码示例展示了 MultiContract 是 SubContract 合约的直接父合约，后跟 DivideContract 和 SumContract：

```
pragma solidity 0.4.19;

contract SumContract {

    function Sum(uint a, uint b) public returns (uint) {
        return  a + b;
    }

}

contract MultiContract is SumContract  {

    function Multiply(uint a, uint b) public returns (uint) {
        return  a * b;
    }

}

contract DivideContract is SumContract  {

    function Divide(uint a, uint b) public returns (uint) {
        return  a / b;
```

严禁网络传播本PDF，违者责任自负！


```
    }
}

contract SubContract is SumContract, MultiContract,DivideContract

    function sub(uint a, uint b) public returns (uint) {
        return  a - b;
    }
}

contract client  {

    function workWithInheritance() public  returns (uint) {
        uint a = 20;
        uint b = 10;
        SubContract subt = new SubContract();
        return subt.Sum(a,b);
    }
}
```

也可以使用合约名称跟随函数名称来调用特定合约中的函数。

6.7 封装

封装是面向对象编程最重要的支柱之一。封装是指对改变其状态的处理，允许隐藏或直接访问状态变量。它指的是声明变量的模式，客户端无法直接访问这些变量，只能使用函数进行修改。这有助于约束对变量的访问，但同时允许对类进行足够的访问以对其进行操作。Solidity 提供多个可见性修改器，例如 `external`、`public`、`internal` 和 `private`，它们影响合约中的状态变量的可见性，继承子合约或外部合约。

6.8 多态性

多态性意味着具有多种形式。有两种类型的多态性：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

- 函数多态性
- 合约多态性

6.8.1　函数多态性

函数多态性是指在同一个合约中声明多个同名函数或继承具有同名函数的合约。同名函数在参数数据类型或参数个数上有所不同。为了确定多态性的有效函数签名，不考虑返回类型。这也被称为**方法重载**。

下面代码段说明了一个包含两个函数的合约，这两个函数具有相同的名称，但传入参数的数据类型不同。第一个函数，`getVariableData` 接受 `int8` 作为其参数数据类型，而同名的另一个函数接受 `int16` 作为其参数数据类型。相同名称的函数使用数目不同或数据类型不同的传入参数，这种方式是绝对合法的，如以下屏幕截图所示：

```
pragma solidity ^0.4.19;

contract HelloFunctionPolymorphism
{
    function getVariableData(int8 data) public constant returns(int8 output)
    {
        return data;
    }

    function overloadedFunction(int16 data) public constant returns(int16 output)
    {
        return data;
    }
}
```

6.8.2　合约多态性

合约多态性是指当合约通过继承相互关联时，可以使用多个合约实例进行互换。合约多态有助于使用基类合约实例调用派生合约函数。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

让我们在下面显示的代码清单的帮助下理解这个概念。

父合约包含两个函数，SetInteger 和 GetInteger。子合约继承自父合约，并提供自己的 GetInteger 实现。可以使用 ChildContract 变量数据类型创建子合约，也可以使用父合约数据类型创建子合约。多态性允许使用基类类型合约变量来引用父子关系中的任何合约。合约实例将决定调用哪个函数——基类合约或派生合约。

看一下下面的代码片段：

```
ParentContract pc = new ChildContract();
```

上述代码使用父合约类型变量来引用新创建的子合约实例。这就是在 Solidity 中实现合约多态的方法，如以下屏幕截图所示：

```
pragma solidity ^0.4.19;

contract ParentContract {

    uint internal simpleInteger;

    function SetInteger(uint _value) public {
        simpleInteger = _value;
    }

    function GetInteger() public view returns (uint) {
        return 10;
    }
}

contract ChildContract is ParentContract {

    function GetInteger() public view returns (uint) {
        return simpleInteger;
    }
}

contract client {
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
    ParentContract pc = new ChildContract();

    function workWithInheritance() public returns (uint) {
        pc.SetInteger(100);
        return pc.GetInteger();
    }
}
```

6.9　方法覆盖

方法覆盖是指在派生合约中重新定义父合约中具有相同名称和签名的函数。

下面的代码段展示了这一点。父合约包含两个函数 SetInteger 和 GetInteger。子合约继承自父合约，并通过覆盖函数提供自己的 GetInteger 来实现。

现在，即使在使用父合约变量时对子合约调用 GetInteger 函数，也会调用子合约实例的函数。这是因为合约中的所有函数都是虚函数并且基于合约实例。调用派生合约的函数，如下面的屏幕截图所示：

```
pragma solidity ^0.4.19;

contract ParentContract {

    uint internal simpleInteger;

    function SetInteger(uint _value) public {
        simpleInteger =  _value;
    }

    function GetInteger() public view returns (uint) {
        return 10;
    }
}

contract ChildContract is ParentContract  {
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
    function GetInteger() public view returns (uint) {
        return simpleInteger;
    }
}

contract client  {

    ParentContract pc = new ChildContract();

    function workWithInheritance() public returns (uint) {
        pc.SetInteger(100);
        return pc.GetInteger();
    }
}
```

6.10 抽象合约

抽象合约是具有部分函数定义的合约。你无法创建抽象合约的实例。为了使用其函数，抽象合约必须由子合约继承。抽象合约有助于定义合约的结构，任何继承类都必须确保提供它们自己的实现。如果子合约没有提供待完善函数的实现，则无法创建它的实例。函数签名使用分号字符结束。Solidity 没有提供将合约标记为抽象合约的关键字。如果合约具有未实现的函数，则它将成为抽象类。

接下来的屏幕截图展示了抽象合约的实现。`abstractHelloWorld` 合约是一个抽象合约，因为它包含几个没有定义的函数。`GetValue` 和 `SetValue` 是没有实现的函数签名。还有另一种返回常量的方法。`AddaNumber` 的目的是表明在抽象合约中也可以包含实现函数。`abstractHelloWorld` 抽象合约被 `HelloWorld` 合约继承，该合约提供了所有方法的实现。`Client` 合约使用基础合约变量创建 `HelloWorld` 合约的实例，并调用其函数，如以下屏幕截图所示：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
pragma solidity 0.4.19;

contract abstractHelloWorld {
    function  GetValue() public view returns (uint);
    function  SetValue(uint _value) public;

    function AddaNumber(uint _value) public returns (uint){
        return 10;
    }

}


contract HelloWorld is abstractHelloWorld{
    uint private simpleInteger;

    function   GetValue() public view returns (uint) {
        return simpleInteger;
    }

    function  SetValue(uint _value) public {
        simpleInteger =  _value;

    }

    function AddaNumber(uint _value) public returns (uint){
        return simpleInteger + _value;
    }
}


contract client  {

    abstractHelloWorld myObj ;

    function client(){
        myObj = new HelloWorld();
    }

    function   GetSetIntegerValue() public  returns (uint) {
       myObj.SetValue(100);
      return  myObj.AddaNumber(200);
    }

}
```

6.11　接口

接口就像抽象合约，但有些差异。接口不能包含任何定义。它们只能包

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

含函数声明。这意味着接口中的函数不能包含任何代码。它们也被称为纯抽象合约。接口只能包含函数的签名。它也不能包含任何状态变量。它们不能继承其他合约或包含枚举或结构。但是，接口可以继承其他接口。函数签名使用分号字符结束。接口使用 `interface` 关键字声明，后跟标识符。以下代码示例显示了接口的实现。Solidity 提供 `interface` 关键字用于声明接口。`IHelloWorld` 接口定义为包含两个函数签名——`GetValue` 和 `SetValue`。没有包含任何函数的实现。`IHelloWorld` 接口是由 `HelloWorld` 合约实现的。合约意图使用此合约创建一个实例，如以下屏幕截图所示：

```
pragma solidity 0.4.19;

interface IHelloWorld {
    function  GetValue() public view returns (uint);
    function  SetValue(uint _value) public;
}


contract HelloWorld is IHelloWorld{
    uint private simpleInteger;

    function   GetValue() public view returns (uint) {
        return simpleInteger;
    }

    function  SetValue(uint _value) public {
        simpleInteger =  _value;

    }
}


contract client  {

    IHelloWorld myObj ;

    function client(){
         myObj = new HelloWorld();
    }

    function   GetSetIntegerValue() public  returns (uint) {
        myObj.SetValue(100);
       return  myObj.GetValue();
    }

}
```

非卖品，仅供非商业用途或交流学习使用

6.12　本章小结

是时候结束本章的讨论内容了。这是一个重量级的篇章，主要关注智能合约，以不同方式创建实例，以及与其相关的所有重要的面向对象概念，包括继承、多态、抽象和封装。可以在 Solidity 中实现多种类型的继承。我们讨论了简单、多层次、分层和多级继承，以及抽象合约和接口的使用与实现。应该注意的是，在 Solidity 中使用继承，最终只有一个合约而不是多个合约被部署。任何具有父子层次结构的合约都只能使用一个地址。

第 7 章将主要关注合约中的函数。函数是编写有效 Solidity 合约的核心。函数有助于更改合约状态并检索它们。离开函数，任何智能合约都很难变得有意义。函数具有不同的可见范围，可影响其行为的多个属性，并且还可以接受以太币。请继续关注第 7 章的旅程！

非卖品，仅供非商业用途或交流学习使用

第 7 章 Chapter 7

函数、修改器和 fallback 函数

Solidity 正在愈发成熟并且提供高级编程结构，以便用户可以编写更好的智能合约。本章专门介绍一些重要的智能合约结构，例如函数、修改器和 fallback 函数。函数是智能合约中继状态变量之后最重要的元素，它帮助我们在以太坊中创建交易和实现自定义逻辑，本章将对各种类型的函数进行深入讨论。修改器是特殊函数，有助于编写更容易获得和模块化的智能合约。fallback 函数对于基于合约的编程语言来说是特有的概念，在合约中任何已有声明的方法都不匹配时被执行。最后，每个函数都附加了可见性，这会影响外部调用者、其他合约和继承合约的可用性。

7.1 函数输入和输出

如果不接受参数和返回值，那么函数就变得不那么有趣了。通用函数使用参数和返回值。参数可以帮助改变函数执行并提供不同的执行路径。Solidity 允

非卖品，仅供非商业用途或交流学习使用

许你在同一函数内接受多个参数，唯一的条件是它们的标识符命名应该唯一。

以下代码片段显示了多个函数，每个函数都有不同的参数和返回值结构：

1. 第一个函数 `singleIncomingParameter` 接受一个类型为 `int` 的参数 `_data`，并返回一个 `int` 类型的返回值 `_output`。函数签名提供了用于定义传入参数和返回值的结构。函数签名中的 `return` 关键字帮助定义函数的返回类型。在下面的代码片段中，函数代码中的 `return` 关键字自动映射到函数签名中声明的第一个返回类型：

```
function singleIncomingParameter(int _data) returns (int
_output) {
    return _data * 2;
}
```

2. 第二个函数 `multipleIncomingParameter` 接受两个参数 `_data` 和 `_data2`，它们都是 `int` 类型，并返回 `int` 类型的单个返回值 `_output`，如下所示：

```
function multipleIncomingParameter(int _data, int _data2)
returns (int _output) {
     return _data * _data2;
 }
```

3. 第三个函数 `multipleOutgoingParameter` 接受一个类型为 `int` 的参数 `_data`，并返回名为 `square` 和 `half` 的两个返回值，它们都是 `int` 类型。在下面的代码片段中，返回多个参数是 Solidity 独有的，在许多编程语言中都找不到：

```
function multipleOutgoingParameter(int _data) returns (int
square, int half)
 {
     square = _data * _data;
     half = _data /2 ;
 }
```

4. 第四个函数 `multipleOutgoingTuple` 与第三个函数类似。但是，它不是将返回值指定为单独的语句和变量，而是将值作为元组返回。**元组**是由多个变量组成的自定义数据结构，如以下代码段所示：

```
function multipleOutgoingParameter(int _data) returns (int
square, int half)
 {
     square = _data * _data;
     half = _data /2 ;
 }
```

合约的完整代码如下面的屏幕截图所示：

```
pragma solidity ^0.4.19;

contract Parameters {

    function singleIncomingParameter(int _data) returns (int _output) {
        return _data * 2;
    }

    function multipleIncomingParameter(int _data, int _data2) returns (int _output) {
        return _data * _data2;
    }

    function multipleOutgoingParameter(int _data) returns (int square, int half) {
        square = _data * _data;
        half = _data /2 ;
    }

    function multipleOutgoingTuple(int _data) returns (int square, int half) {
        (square, half) = (_data * _data,_data /2 );
    }

}
```

也可以在没有任何标识符的情况下声明参数。但是，此功能没有多大用处，因为这些参数无法在函数代码中引用。同样，可以不使用任何名字来声明返回值。

7.2　修改器

修改器是 Solidity 独有的另一个概念。修改器帮助我们修改函数的行为。

我们试着通过一个例子来理解这一点。以下代码不使用修改器，在这个合约中，定义了两个状态变量、两个函数和一个构造函数。其中一个状态变量存储部署合约的账户地址。在构造函数中，全局变量 `msg.sender` 用于在所有者状态变量中输入账户值。两个函数检查调用者是否与部署合约的账户相同，如果相同，则执行函数代码，否则忽略其余代码。虽然此代码可以正常工作，但在可读性和可管理性方面还可以做得更好。此时修改器可以提供帮助。在此示例中，使用 `if` 条件语句进行检查。在下一章中，我们将看到如何在没有 `if` 条件的情况下使用新的 Solidity 结构（例如 `require` 和 `assert`）执行相同的检查。看一下如下屏幕截图，它描述了修改器的代码片段：

```
pragma solidity ^0.4.17;

contract ContractWithoutModifier {

    address owner;
    int public mydata;

    function ContractWithoutModifier(){
        owner = msg.sender;
    }

    function AssignDoubleValue(int _data) public  {
        if(msg.sender == owner) {
            mydata = _data * 2;
        }
    }

    function AssignTenerValue(int _data) public  {
        if(msg.sender == owner) {
            mydata = _data * 10;
        }
    }

}
```

修改器是改变函数行为的特殊函数。这里，函数代码保持不变，但函数的执行路径发生了变化。修改器只能应用于函数。现在让我们看看如何使用修改器编写相同的合约，如以下屏幕截图所示：

```
pragma solidity ^0.4.17;

contract ContractWithModifier {
```

```
    address owner;
    int public mydata;

    function ContractWithModifier(){
        owner = msg.sender;
    }

    modifier isOwner {
        // require(msg.sender == owner);
        if(msg.sender == owner) {
            _;
        }
    }

    function AssignDoubleValue(int _data) public isOwner {
            mydata = _data * 2;
    }

    function AssignTenerValue(int _data) public  {
            mydata = _data * 10;
    }

}
```

此处显示的合约具有相同的结构：一个构造函数、两个状态变量和两个函数。它还有一个附加的使用 `modifier` 关键字定义的特殊函数。`AssignDoubleValue` 和 `AssignTenerValue` 函数的代码是不同的，尽管它们具有类似的功能。这两个函数不使用 `if` 条件来检查函数的调用者是否与部署合约的账户相同，而是在其签名中使用修改器名称进行修饰。

现在让我们尝试理解 Solidity 中修改器的结构及其用法。

使用关键字 `modifier` 和标识符定义修改器。修改器的代码放在大括号内。修改器中的代码可以验证传入的值，并且可以在评估后有条件地执行被调用的函数。在这里标识符“`_`”特别重要——它的目的是用调用的函数代码替换自己。

当调用用修改器 `isOwner` 修饰的 `AssignDoubleValue` 函数时，修改器控制执行并将其标识符替换为调用函数 `AssignDoubleValue` 的代码。最终，在 EVM 中，运行时修改器看起来像下面的代码：

```
modifier isOwner {
// require(msg.sender == owner);
if(msg.sender == owner) {
mydata = _data * 2;
}
}
```

相同的修改器可以应用于多个函数，_标识符可以替换为被调用的函数代码。

这有助于编写更清晰、更易读、更易维护的代码。开发人员不必在每个函数中重复相同的代码，也不必在执行函数时检查传入的值。

7.3 view 函数、constant 函数和 pure 函数

Solidity 为函数提供了特殊修改器，例如 `view`、`pure` 和 `constant`。这些也称为**状态可变**属性，因为它们定义了以太坊全局状态中允许修改的范围。这些修改器除了存在一些小的差异外，与前面讨论过的修改器的目的相似。本节将详细介绍这些关键字的用法。

编写智能合约函数主要有以下三项活动：

- 更新状态变量
- 读取状态变量
- 逻辑执行

函数的执行和交易并不是免费的，需要花费 gas。每笔交易都需要花费与其执行相对应的一定量的 gas，调用者需要负责提供足量的 gas 以便成功执行。交易或修改以太坊全局状态的任何活动都如此。

有些函数仅负责读取和返回状态变量，这些函数就像其他编程语言中的 getter 属性一样。它们读取当前状态变量的值并将其返回给调用者。这些函数不

会改变以太坊的状态。以太坊的文档（http://solidity.readthedocs.io/en/v0.4.21/contracts.html）提到以下与修改状态相关的语句：

- 写状态变量
- 发出事件
- 创建其他合约
- 使用 selfdestruct
- 通过调用发送以太币
- 调用任何未标记为 view 或 pure 的函数
- 使用低级调用
- 使用包含某些操作码的内联汇编

Solidity 开发人员可以使用 view 修改器标记其函数，以建议 EVM 此函数不会更改以太坊状态或之前提及的任何活动。目前，这不是强制执行的，但预计未来会实施。

以下屏幕截图显示了 view 函数的示例：

```
pragma solidity ^0.4.17;

contract ViewFunction {

    function GetTenerValue(int _data) public view returns (int)  {
            return _data * 10;
    }

}
```

如果函数没有修改任何状态只返回值，则可以使用 view 标记它。

值得注意的是，view 函数也称为**常量**函数。常量函数用于先前版本的 Solidity。

严禁网络传播本PDF，违者责任自负！

与 view 函数相比，pure 函数在状态可变性方面有更多限制。但是，它们的目的是相同的，即限制状态的可变性。值得注意的是，即使是 pure 函数也没有在编写代码时强制要求，但预计将来会实施。

pure 函数在 view 函数之上添加了进一步的限制，例如，甚至不允许 pure 函数读取以太坊的当前状态。简而言之，不允许 pure 函数读写以太坊的全局状态。根据文档其他不允许的活动包括：

- 读取状态变量
- 访问 this.balance 或 <address>.balance
- 访问 block、tx 和 msg 的任何成员（msg.sig 和 msg.data 除外）
- 调用任何未标记为 pure 的函数
- 使用包含某些操作码的内联汇编

之前的函数使用 pure 函数重写后如以下屏幕截图所示：

```
pragma solidity ^0.4.17;

contract PureFunction {

    function GetTenerValue(int _data) public pure returns (int)  {
            return _data * 10;
    }

}
```

7.4 地址相关函数

在与数据类型相关的章节中，我们故意没有解释与 address 数据类型相关的函数。虽然可以在那里包括这些函数，但其中一些函数可以自动执行 fallback 函数，因此在这里进行介绍。

严禁网络传播本PDF，违者责任自负！

地址提供了五个函数和一个属性。

地址提供的唯一属性是 balance 属性，它提供以 wei 为单位计算的账户（合约或个人）中可用的余额，如以下代码段所示：

```
<<account>>.balance ;
```

在上面的代码中，account 是一个有效的以太坊地址，它将返回以 wei 为单位计算的余额。

现在，让我们来看看账户提供的方法。

7.4.1　send 方法

使用 send 方法将以太币发送到合约或个人拥有的账户。看一下如下描述的 send 方法的代码：

```
<<account>>.send(amount);
```

send 函数提供了不能被取代的 2 300 个 gas 的固定费用。将金额发送到合约地址时，这点很重要。要将金额发送到个人拥有的账户，这个数量的 gas 就足够了。send 函数返回一个布尔值 true/false。在这种情况下，不返回异常。

相反，函数返回 false。如果一切都正确执行，则函数返回 true。如果 send 与合约地址一起使用，它将调用合约的 fallback 函数。我们将在后面详细研究 fallback 函数。

现在，让我们来看一下 send 函数的示例，如以下屏幕截图所示：

严禁网络传播本PDF，违者责任自负！


```
function SimpleSendToAccount() public returns (bool) {
    return msg.sender.send(1);
}
```

在上面的屏幕截图中，send 函数将 1wei 发送给 SimpleSendToAccount 函数的调用者。在前面的章节中，我们已经了解了 msg.sender 是处理全局变量的。

send 是一个低级函数，应该谨慎使用，因为它可以调用 fallback 函数，可以在合约中一次又一次地递归调用。有一种称为 Check-Deduct-Transfer（CDF）的模式，有时也称为 Check-Effects-Interaction（CEI），我们将在下面的屏幕截图中看到。在此模式中，假设在映射中保存余额。映射由地址及其关联的余额组成，如以下屏幕截图所示：

```
mapping (address => uint) balance;

function SimpleSendToAccount(uint amount) public returns (bool) {
    if(balance[msg.sender] >= amount ) {
        balance[msg.sender] -= amount;
        if (msg.sender.send(amount) == true) {
            return true;
        }
        else {
            balance[msg.sender] += amount;
            return false;
        }
    }
}
```

在此示例中，首先检查调用者是否具有足够的余额来提取资金。如果有，我们将现有余额减去指定数量的金额并调用 send 方法。然后，必须检查发送是否成功；如果没有，退还金额。

值得注意的是，许多消息来源声称 send 已被弃用，但我不这么认为。send 函数仍有特定用途，例如向多个账户发送金额。尽管引入了新的 transfer 函数用于从一个账户发送以太币到另一个账户，但要求其他合约和账户调用特定方法来提取金额应该是更好的解决方案。

严禁网络传播本PDF，违者责任自负！

7.4.2　transfer 方法

transfer 方法类似于 send 方法，它负责将以太币或 wei 发送到一个地址。但是，区别在于，在执行失败的情况下，transfer 会引发异常，而不是返回 false，并且所有更改都将被还原。看一下以下屏幕截图中的 transfer 方法：

```
function SimpleTransferToAccount() public  {
    msg.sender.transfer(1);
}
```

transfer 方法优于 send 方法，因为它在发生错误时引发异常，这意味着在堆栈中产生异常并停止执行。

7.4.3　call 方法

call 方法使许多开发人员感到困惑。有一个可用的 web3.eth 对象的 call 方法，还有 <<address>>.call 函数。这是两个具有不同目的的函数。

web3.eth 的 call 方法只能调用它所连接的节点，并且是只读操作。不允许改变以太坊的状态。它不会产生交易，也不消耗任何 gas。它用于调用 pure 函数、constant 函数和 view 函数。

另一方面，地址数据类型提供的 call 函数可以调用合约中任何可用的函数。有时候合约接口（通常称为 ABI）不可用，因此调用函数的唯一方法是使用 call 方法。此方法不依附于 ABI，可以根据需要调用任何函数。这些调用没有编译时检查，返回布尔值 true 或 false。

值得注意的是，使用 call 方法调用合约函数不是理想的做法，因为不涉及检查和验证。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

合约中的每个函数都在运行时使用 4 字节标识符进行标识。这个 4 字节的标识符是函数名称及其参数类型的精简散列。在对函数名称和参数类型进行散列后，前四个字节被视为函数标识符。call 函数接受这些字节以将函数作为第一个参数调用，将实际参数值作为后续参数调用。

以下代码中显示了没有任何参数的调用函数。这里，SetBalance 不带任何参数：

```
myaddr.call(bytes4(sha3("SetBalance()")));
```

以下代码中显示了带参数的调用函数。这里，SetBalance 带了一个 unint 参数：

```
myaddr.call(bytes4(sha3("SetBalance(uint256)")), 10);
```

值得注意的是，之前看到的 send 函数实际上内部调用 call 函数，并不需要提供 gas。以下代码示例显示了所有可能使用此函数的方法。在本例中，使用以下两个简单函数创建名为 EtherBox 的合约：

- SetBalance：它有一个状态变量，这个函数的目的是在每次调用时将状态变量的现有值加 10。
- GetBalance：此函数负责返回状态变量的当前值。

创建另一个名为 usingCall 的合约，以通过 call 函数调用 EtherBox 合约上的方法。让我们看一下如下代码示例中提到的函数：

1. SimpleCall：此函数创建 EtherBox 合约的实例并将其转换为地址。使用这个地址，call 函数调用 EtherBox 合约中的 SetBalance 函数。
2. SimpleCallWithGas：此函数创建 EtherBox 合约的实例并将其转

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

换为地址。使用这个地址，call 函数调用 EtherBox 合约中的 SetBalance 函数。如果函数的执行需要更多 gas，跟随 call 函数一起发送 gas。

3. SimpleCallWithGasAndValue：此函数创建 EtherBox 合约的实例并将其转换为地址。使用这个地址，call 函数调用 EtherBox 合约中的 SetBalance 函数。如果函数的执行需要更多 gas，跟随 call 函数一起发送 gas。除了 gas 之外，还可以将向支付函数发送以太币或 wei（以太币最小货币单位）。

看一下如下屏幕截图中前面提到的函数：

```
pragma solidity ^0.4.17;

contract EtherBox {
    uint balance;

    function SetBalance() public {
        balance = balance + 10;
    }

    function GetBalance() public payable returns(uint) {
       return balance;
    }
}

contract UsingCall {
    function UsingCall() public payable  {
    }

    function SimpleCall() public returns (uint) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        status =   myaddr.call(bytes4(sha3("SetBalance()")));
        return eb.GetBalance();
    }

    function SimpleCallwithGas() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        status =   myaddr.call.gas(200000)(bytes4(sha3("GetBalance()")));
        return status;
    }
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
    function SimpleCallwithGasAndValue() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        status =    myaddr.call.gas(200000).value(1)(bytes4(sha3("GetBalance()")
        return status;
    }

}
```

7.4.4 callcode 方法

此函数已弃用，此处不再讨论。有关 callcode 的更多信息，请访问 https://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html。

7.4.5 delegatecall 方法

此函数同样是一个低级函数，负责使用调用者的状态变量调用另一个合约中的函数。通常，它与 Solidity 中的库一起使用。有关 delegatecall 的更多信息，请访问 https://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html。

7.5 fallback 函数

fallback 函数是一种仅在以太坊中可用的特殊类型的函数。Solidity 支持编写 fallback 函数。想象一下，作为 Solidity 开发人员，你通过调用智能合约中的函数来进行消费。你很可能使用该合约中不存在的函数名称。在这种情况下，顾名思义，fallback 函数将自动被调用。

当没有函数名称与被调用函数匹配时，将调用 fallback 函数。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

fallback 函数没有标识符或函数名称。它是定义的无名函数。由于无法显式调用，因此无法接受任何参数或返回任何值。fallback 函数的示例如下：

```
pragma solidity ^0.4.17;

contract FallbackFunction {

    function () {
        var a = 0;
    }

}
```

当合约收到任何以太币时，也可以调用 fallback 函数。这通常发生在使用 web3 中提供的 SendTransaction 函数将以太币从一个账户发送给合约的情况下。但是，在这种情况下，fallback 函数应该标记为 payable，否则它将无法接受以太币并且将引发错误。

需要回答的下一个重要问题是执行此函数需要多少 gas。由于无法明确调用，因此无法将 gas 发送到此函数。相反，EVM 为此功能提供了 2 300 个 gas 的固定费用。任何超过此限制的 gas 消耗都会引发异常，并且在消耗了与原始函数一起发送的所有 gas 后，状态将被回滚。因此，重要的是测试你的 fallback 函数，以确保它不会消耗超过 2 300 个 gas。

值得注意的是，fallback 函数是智能合约中安全漏洞的主要原因之一。在发布生产合约之前，从安全角度测试此函数是非常重要的。

现在让我们尝试通过一些例子理解 fallback 函数。

我们将使用与解释地址数据类型的调用函数相同的示例。但是，这一次，我们在 EtherBox 合约中实现了一个标记为 payable 的 fallback 函数，其目的是引发一个事件和一个调用无效函数的附加函数。该事件也在函数内声明。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

我们将在下一章更深入地研究事件。

当你执行 UsingCall 合约中的每个方法时，应该注意到调用正确函数的函数，不会触发调用 fallback 函数，如以下屏幕截图所示：

```
pragma solidity ^0.4.17;

contract EtherBox {
    uint balance;
    event logme(string);

    function SetBalance() public {
        balance = balance + 10;
    }

    function GetBalance() public payable returns(uint) {
       return balance;
    }

    function() payable {
        logme("fallback called");
    }
}

contract UsingCall {
    function UsingCall() public payable  {
    }

    function SimpleCall() public returns (uint) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        status =   myaddr.call(bytes4(sha3("SetBalance()")));
        return eb.GetBalance();
    }

    function SimpleCallwithGas() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        return status =   myaddr.call.gas(200000)(bytes4(sha3("GetBalance()")));
    }

    function SimpleCallwithGasAndValue() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        return status =   myaddr.call.gas(200000).value(1)(bytes4(sha3("GetBalance()")));
    }

    function SimpleCallwithGasAndValueWithWrongName() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        return myaddr.call.gas(200000).value(1)(bytes4(sha3("GetBalance1()")));
    }
}
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

使用 send 方法，使用 web3SendTransaction 函数或 transfer 方法时也会 fallback 函数。

7.6　本章小结

再次强调一下，本章是很重要的章节，主要关注函数，包括 address 函数以及 pure 函数、constant 函数和 view 函数。address 函数可能令人生畏，尤其是当你考虑它们的多种变化及其与 fallback 函数的关系时。如果要实现 fallback 函数，请记住要特别注意测试，尤其是从安全角度来看。当使用低级 Solidity 函数如 send、call 和 transfer 时，还应特别注意它们会隐式调用 fallback 函数。始终尝试使用 ABI 的合约函数，因为它确保调用正确的函数及其数据类型。

在第 8 章中，我们将深入探讨 Solidity 中的事件、日志和异常处理。

非卖品，仅供非商业用途或交流学习使用

第 8 章 Chapter 8

异常、事件与日志

使用 Solidity 最基本的作用就是编写智能合约。而编写智能合约需要健壮的错误和异常处理。错误和异常处理是编程中的常态，Solidity 也为二者提供了充分的基础架构层面的支持。编写健壮智能合约的最佳实践之一就是要有恰如其分的错误和异常处理。事件是 Solidity 的另外一个重要组成部分。到目前为止在我们所讨论过的所有主题中，我们都见识了调用者调用合约中的方法，然而我们并没有讨论智能合约通过何种机制去通知调用者和其他人关于合约状态或其他信息的变化。这就是事件起作用的地方。事件是事件驱动编程的一部分，根据程序内部的变化，主动向调用者通知这种变化。调用者可以自由选择是使用这个通知信息还是忽略它。在更大层面来讲，异常和事件都是应用了以太坊虚拟机（EVM）提供的日志处理功能。

非卖品，仅供非商业用途或交流学习使用

8.1 错误处理

编写智能合约经常会在不经意间引入一些错误，因此编写健壮的合约是一个应该遵循的实践法则。在编程世界错误是生活的一部分，编写完全无差错的合约是一种值得追求的理想技能。错误有可能在设计阶段也可能在运行阶段发生。由于 Solidity 会被编译成字节码，在编译的过程中会对设计阶段产生的各种语法错误进行检查。然而，对于运行时错误来说，它们通常在运行时发生并且更加难以捕获。针对各种可能的运行时错误对合约进行测试是很重要的，但更重要的是写出对两种错误都有防御性的健壮性合约。

常见的运行时错误有 gas 耗尽、零除错误、数据类型溢出、数组越界等。

在 4.10 版本之前，Solidity 只有一个 `throw` 语句进行错误处理。开发者需要写多个 `if...else` 语句去检查程序中的数值并在碰到错误情况时进行 `throw` 处理。这个 `throw` 语句会消耗掉所有提供的 gas 并将状态回退到初始状态。这种处理方式对于架构师和开发人员来说并不理想，未使用的 gas 本应该退回给合约调用者。

从 4.10 版本开始，一些新的错误处理结构比如 `assert`、`require` 和 `revert` 等被引入，`throw` 语句也被标记为过时。在这一部分，我们深入探讨这些新的错误处理结构。值得注意的是在 Solidity 中并没有 `try...catch` 这种语句或者说结构去捕获错误和异常。

8.1.1 require 语句

`require` 这个单词表示的是种约束。声明 `require` 语句意味着声明运行该函数的前提条件。换句话说，这意味着在运行接下来的代码之前，所声明的约束必须首先被满足。

非卖品，仅供非商业用途或交流学习使用

require 语句接受一个参数：一个执行结果为 true 或 false 的语句。如果该语句的执行结果为 false，异常会被抛出并且程序执行会被挂起，未消耗的 gas 会被退回给合约调用者并且合约状态会被回退到初始状态。require 语句导致 revert 指令的执行，用来回退合约状态并退回未消耗的 gas。

下面的代码展示了 require 语句的使用。

```
pragma solidity ^0.4.19;

contract RequireContract {

    function ValidInt8(uint _data) public returns(uint8){
        require(_data >= 0);
        require(_data <= 255);

        return uint8(_data);
    }

    function ShouldbeEven(uint _data) public returns(bool){
        require(_data % 2 == 0);
        return true;
    }

}
```

我们来看一下上面截图中展示的两个函数：

ValidInt8：这个函数用了好几个 require 语句。在代码块里面，一个语句用来检查传入值是否大于或等于 0。如果该语句结果为 true，继续往下执行下一个语句。如果结果为 false，则抛出异常并终止程序执行。下一个语句是检查传入值是否小于或等于 255。如果大于 255，改语句结果就为 false，异常会被抛出。

ShouldbeEven：这个函数也是类似的。在这个函数中，require 语句用来检查传入参数值是奇数还是偶数。如果传入值为偶数，继续往下执行下一个语句，否则抛出异常。

require 语句应该用于校验所有传入函数的参数或数值。这意味着不管是另外一个合约的函数还是本合约的函数，传入的值都应该用 require 语句进行校验。这个 require 语句应该在变量使用前检查其当前状态。如果有异常抛出，则说明传给该函数的值不符合函数期望，调用者应该在将这个参数值发给合约之前对其进行相应的调整。

8.1.2 assert 语句

assert 语句和 require 有类似的语法。它接受一个结果为 true 或 false 的语句。基于这个结果来确定是继续执行还是抛出异常。并且未消耗的 gas 不会被退回给调用者而是被 assert 语句完全消费掉，合约会被回退到初始状态。Assert 语句会产生一个 invalid 指令用来负责回退状态与耗尽 gas。

我们对前面提到的函数进行扩展，对现有变量引入了加法操作。然而，对两个变量进行加法操作是有可能导致溢出异常的。这里可以用 assert 语句进行校验，如果校验结果为 true，返回变量相加结果，否则抛出异常。

下面这张截图展示了 assert 语句的使用：

```
pragma solidity ^0.4.19;

contract AssertContract {

    function ValidInt8(uint _data) public returns(uint8){
        require(_data >= 0);
        require(_data <= 255);

        uint8 value = 20;

        //checking datatype overflow
        assert (value + _data <= 255);

        return uint8(value + _data);
    }
}
```

当 require 被用来校验外部传入的值时，assert 应该被用来在合约和函数执行前验证其当前状态与条件。可以把 assert 当作是专门用来处理无法预测的运行时异常的。你应该在认为当前状态已经变得不一致而不适合继续运行时使用 assert。

8.1.3　revert 语句

revert 语句与 require 的作用是非常相似的。不同的是，它并不去检查条件也不依赖于任何状态或条件。碰到 revert 语句意味着异常会被抛出，未消耗的 gas 被退回，同时合约状态被恢复。

在下面的例子中，在用 if 进行条件检查时抛出异常。如果 if 条件的结果为 false 就会执行 revert，这会导致异常的发生和合约执行的终止。

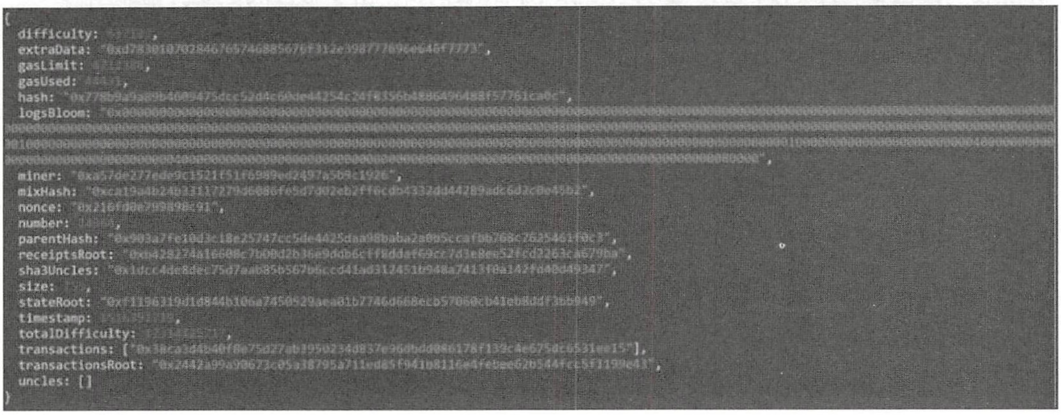
```
pragma solidity ^0.4.19;

contract RevertContract {

    function ValidInt8(int _data) public returns(uint8){

        if(_data < 0 || _data > 255) {
            revert();
        }

        return uint8(_data);
    }
}
```

8.2　事件与日志

我们已经在前面的章节中看到了事件的用法，但没有详细说明。但是，在本节中，我们将更深入地研究事件。对于做事件驱动相关开发的程序员来说事件是众人皆知的概念。事件是指合约中的某些更改，这些更改会引发事件发送并与事件监听者相互通知，以便它们可以行动起来并执行其他功能。

事件可以帮助我们写出异步应用程序。除了不断轮询以太坊账本以确定某个交易是否存在，然后为了获取某些信息而进行阻塞，还可以使用事件来实现同样的事情。这样，以太坊平台将通知客户是否有事件发生。这有助于编写模块化代码并节省资源。

事件是合约继承的一部分，子合约可以调用事件。事件数据与区块数据一起存储。`logsBloom` 值是事件数据，如以下屏幕截图所示：



在 Solidity 中声明事件与执行函数非常相似。但是，事件没有任何代码体。一个简单的事件可以使用 `event` 关键字后跟一个标识符以及它想要随事件一起发送的任何参数来声明，如以下代码所示：

```
event LogFunctionFlow(string);
```

在前面的代码行中，`event` 是用于声明事件的关键字，后面跟着它的名称和一组将随该事件一起发送的参数。任何字符串文本都可以使用 `LogFunctionFlow` 事件发送。

使用事件很简单，只需使用其名称调用事件并传递它所期望的参数即可。

对于 LogFunctionFlow 事件，调用将如下所示，类似于带参数的函数调用：

```
LogFunctionFlow("I am within function x");
```

以下代码片段显示了正在使用的事件。在这个例子中，一个事件 LogFunctionFlow 以字符串作为唯一参数声明。ValidInt8 函数会多次调用同一个事件，在函数的各个阶段提供文本信息：

```
pragma solidity ^0.4.19;

contract EventContract {

    event LogFunctionFlow(string);

    function ValidInt8(int _data) public returns(uint8){
        LogFunctionFlow("Within function ValidInt8");

        if(_data < 0 || _data > 255) {
            revert();
        }

        LogFunctionFlow("Value is within expected range");
        LogFunctionFlow("Returning value from function");

        return uint8(_data);
    }
}
```

在 Remix 中执行此合约会显示事件结果，其中包含三个带有事件信息的日志，如以下屏幕截图所示：

```
logs    [
            {
                    "topic": "b5b850034705238ab6360bdc803e9e3dcaaf926c812b20193e2e99a5918d47b0",
                    "event": "LogFunctionFlow",
                    "args": [
                            "Within function ValidInt8"
                    ]
            },
            {
                    "topic": "b5b850034705238ab6360bdc803e9e3dcaaf926c812b20193e2e99a5918d47b0",
                    "event": "LogFunctionFlow",
                    "args": [
                            "Value is within expected range"
                    ]
            },
            {
```

```
logs                    "topic": "b5b850034705238ab6360bdc803e9e3dcaaf926c812b20193e2e99a5918d47b0",
                        "event": "LogFunctionFlow",
                        "args": [
                                "Returning value from function"
                        ]
                }
        ]
```

还可以使用 web3 从自定义应用和去中心化应用中监听事件。

注意：事件可以使用参数名称进行过滤。

以下两种方法可以让我们监听事件：

1. **监听单个事件：**在这种使用 web3 的方法中，可以监听和跟踪合约中的单个事件。当有确切事件是从合约中触发的时候，它有助于在 web3 客户端执行一个函数功能。以下屏幕截图显示了监听单个事件的示例。在这里，ageRead 是我们感兴趣并监听的事件的名称。我们从块编号 25000 读取，直到最新块。首先，创建一个到 ageRead 事件的引用并将监听者添加到这个引用中。监听者需要一个在 ageRead 事件触发时执行的 promise 函数：

```
var myEvent = instance.ageRead({fromBlock: 25000, toBlock: 'latest'});
myEvent.watch(function(error, result){
    if(error) {
        console.log(error);
    }
    console.log(result.args)
});
```

2. **监听所有事件：**在这种使用 web3 的方法中，可以监听和跟踪来自合约的所有事件。当合约触发任何事件时，它会通知并帮助在 web3 客户端执行一项功能作为回应。在这种情况下，可以使用事件名称过滤事件。以下屏幕截图显示了监听所有事件的示例。在这里，我们对合约中的任何事件感兴趣并进行监听。我们从编号 25000 的区块开始读取，直到最新块。首先，创建一

个对 allEvents 的引用并将一个监听者添加到引用中。监听者随后接受一个 promise 函数，该函数在有任何事件被触发时执行：

```
var myEvent = instance.allEvents({fromBlock: 24000, toBlock: 'latest'});
myEvent.watch(function(error, result){
    if(error) {
        console.log(error);
    }
    console.log(result)
});
```

以下屏幕截图显示了事件结果对象中的值：

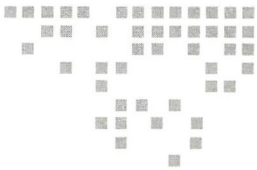
```
{ address: '0x600c320dd768fb55f03748d4d4028db2cafc06a9',
  blockNumber: 24864,
  transactionHash: '0x38ca3d4b40f8e75d27ab3950234d837e96dbdd086178f139c4e675dc6531ee15',
  transactionIndex: 0,
  blockHash: '0x778b9a9a89b4609475dcc52d4c60de44254c24f8356b4886496488f57761ca0c',
  logIndex: 0,
  removed: false,
  event: 'ageRead',
  args: { '': '33' } }
```

8.3　本章小结

在本章中，我们介绍了异常处理和事件。这些都是 Solidity 中的重要主题，特别是在以太坊平台上撰写重要的去中心化应用的时候。Solidity 中的异常处理赖于三个函数实现：assert、require 和 revert。尽管它们听起来很相似，但却有着不同的用途，本章通过实例进行了讲解。事件帮助我们编写具有可扩展性的应用。与其持续不断地轮询平台去获取数据进而导致资源的浪费，我们最好先编写事件，然后等待事件的处罚去异步执行函数，本章也对此进行了介绍。

在第 9 章中，我们将重点介绍 Truffle 的使用，这是在以太坊平台上开发应用最流行的开发平台之一。敬请关注！

严禁网络传播本PDF，违者责任自负！



第 9 章 Chapter 9

Truffle 基础与单元测试

编程语言需要丰富的工具生态来简化开发。与任何应用程序一样，即使是基于区块链的去中心化应用也应该具有最简化的**应用程序生命周期管理**（ALM）过程。对于任何应用程序来说，持续进行构建、测试和部署都非常重要。Solidity 是一种编程语言，需要其他工具的支持，以确保开发人员能够轻松地开发、构建、测试和部署智能合约，而不是困难重重地进行这些工作。这提高了它们的生产力，并最终有助于将应用程序更快、更好、更便宜地推向市场。在这些工具的帮助下，还可以引入用于智能合约的 DevOps 过程。Truffle 就是这样的一种工具，它可以使开发、测试和部署这些活动变得轻而易举。

9.1 应用程序开发生命周期管理

如前所述，每个严肃的应用程序都有相应的开发过程。通常，它涉及设计、构建、测试和部署。智能合约的开发生命周期和其他软件或程序并没什么不同。

严禁网络传播本PDF，违者责任自负！

第一步就是收集并确定所考虑问题的需求，需求获取是任何去中心化应用程序的起始活动。需求包含问题描述、用例和详细测试策略。架构师们将功能性需求和技术性需求作为它们的输入，来进行应用程序的架构与设计。

他们还使用其他人容易理解的符号来记录它们。项目开发团队将这些架构和设计文档分解为 feature 和 sprint。然后根据本文档开始构建智能合约和其他组件。合约经常被部署到测试环境进行测试，以确保它们在技术和功能上都处于正常工作状态。我们通过单元测试来独立检查合约的功能。如果有单元测试失败，则重复整个构建和测试过程。最后，所有组件都被部署到生产环境中。

正如你所看到的，ALM 是一个过程并且会耗费开发人员大量的时间和精力，需要工具和自动化的方式来帮助缓解这一过程，而这正是 Truffle 作为实用工具令人兴奋的地方。

9.2 Truffle

Truffle 是一种加速器，可帮助提高开发、部署和测试的速度，并提高开发人员的生产力。它专为基于以太坊的合约和应用程序开发而构建。最新的 Truffle 版本为 4。它是一个基于 node 运行时的框架，可以帮助轻松实施 DevOps，持续集成、持续交付和持续部署。

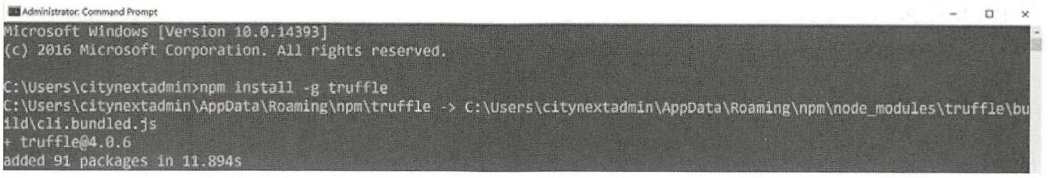
安装 Truffle 非常简单——安装 Truffle 的先决条件是 Node.js，因为它作为 node 包进行部署。

可以通过从命令行执行以下 npm 命令来安装 Truffle：

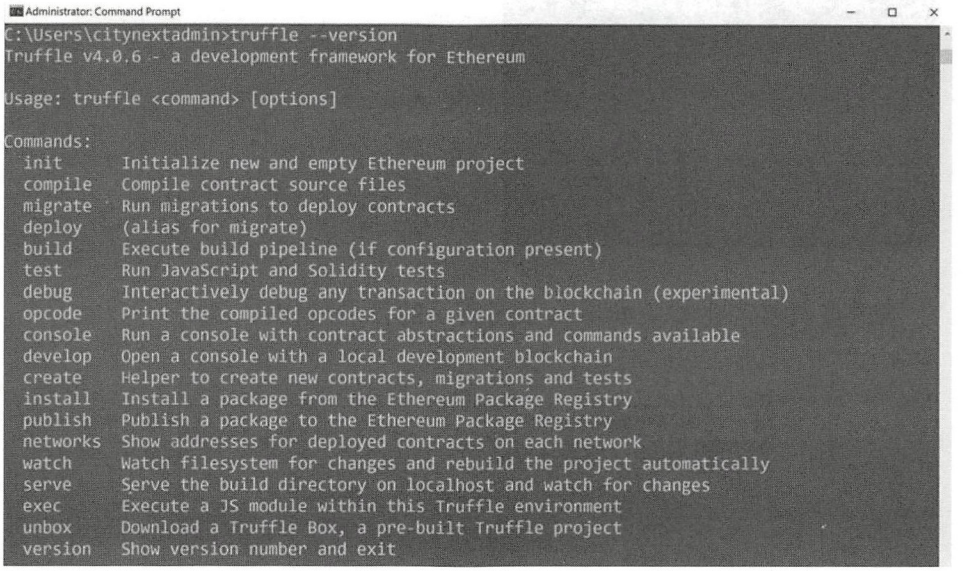
```
$ npm install -g truffle
```

严禁网络传播本PDF，违者责任自负！

这里 npm 是指 node 包管理器，-g 开关表示在全局范围内安装。下面的屏幕截图显示了在 WindowsServer2016 上安装 Truffle 的情况。同样的命令也适用于 Linux 发行版：



运行 truffle--version 显示当前版本和 Truffle 提供的所有命令，如以下屏幕截图所示：



9.3　使用 Truffle 进行开发

使用 Truffle 非常简单。Truffle 在默认情况下提供大量的脚手架代码和配置。

严禁网络传播本PDF，违者责任自负！

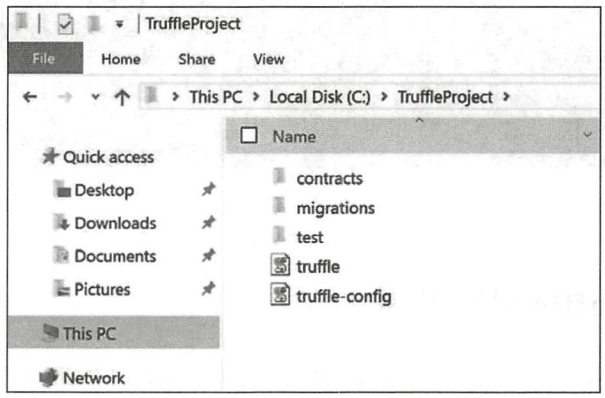
开发人员只需要重新配置一些开箱即用的配置选项，并专注于编写合约。我们来看看以下使用步骤：

1. 第一步是创建一个项目目录，该目录将包含所有该项目相关和 Truffle 生成的工件。

2. 转到该目录并输入 `init` 命令。`init` 命令用于 Truffle 的启动和初始化。它将在该目录中生成适当的子目录、代码文件、配置和链接，如以下屏幕截图所示：

```
C:\>mkdir TruffleProject

C:\>cd TruffleProject

C:\TruffleProject>truffle init
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
```

上面的代码将产生如下面屏幕截图所示的目录结构：



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

让我们来看看上面屏幕截图中显示的以下目录：

- `contracts` 目录包含一个名为 `migrations.sol` 的文件。该文件包含一个负责将自定义合约部署到以太坊网络上的合约。任何自定义合约都应放置在此目录中。

`migrations` 目录包含多个用于执行合约部署过程的 JavaScript 文件。这些 JavaScript 文件应该被修改以确保所有自定义合约对 Truffle 可见，Truffle 可以按照适当的顺序对合约进行链接和部署。这些 JavaScript 脚本以数字为前缀并以从 1 为前缀的脚本开始连续顺序执行。

- `test` 目录是空的，但是任何自定义测试脚本都应放到该目录中。
- 还有两个 JSON 格式的配置文件—`truffle` 和 `truffle-config`。对一个项目来说最主要的配置文件是 `truffle.js`，该文件应该针对每个项目进行定制。它会导出一个 JSON 对象，以便于 Truffle 在运行时使用其进行环境配置。
- 此处应提供的一个重要配置是网络信息，Truffle 基于此将合约部署到相应网络中。

3. 下面的代码片段可用来进行网络配置。应该有一个正在运行并打开 RPC 端口的 Geth 实例，也可以使用 ganache-cli 来代替 geth 通过 JSON-RPC 协议来部署合约。这里的网络配置元素应该被定义以便于连接到现有的以太坊网络。每个网络可以指定一个名称，并且为不同的环境配置不同的网络：

```
module.exports = {
    networks: {
      development: {
          host: "127.0.0.1",
          port: 8545,
          network_id: "*" // Match any network id
      }
   }
};
```

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

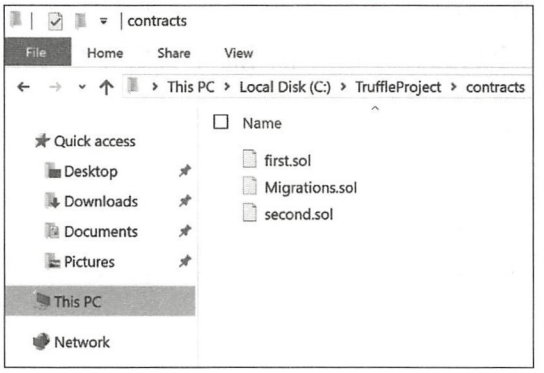
4. 创建一个名为 first.sol 的合约文件并将其保存在 contracts 目录中，如下面屏幕截图所示：

```
pragma solidity ^0.4.17;
contract First {
        int public mydata;
        function GetDouble(int _data) public returns (int
_output) {
                    mydata = _data * 2;
                    return _data * 2;
         }
     }
```

5. 编写另外一个如下面屏幕截图所示的名为 second.sol 的合约并将其放到同一个目录中。

```
pragma solidity ^0.4.17;
import "./first.sol";
         contract Second {
                address firstAddress;
                int public _data;
               function Second(address _first) public {
                   firstAddress = _first;
               }
              function SetData() public {
                 First h = First(firstAddress);
                _data = h.GetDouble(21);
             }
       }
```

最终，contracts 目录看起来会如下面截图一样：



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

6. 修改 migrations 目录，将另外一个脚本文件添加到其中。值得注意的是，这里的每个文件都应该根据合约部署顺序将其前缀加 1。在我们这种情况下，新添加的文件名为 2_Custome.js。文件的内容下面会展示出来。文件的头两行引用了前面所写的两个合约，并暴露了一个可以在合约部署时被 Truffle 调用的一个函数。这个函数先是部署第一个合约，在第一个合约被成功部署以后，再部署第二个合约。如下面的屏幕截图所示：

```
var hw = artifacts.require("First");

var hw1 = artifacts.require("Second");

module.exports = function(deployer) {
  deployer.deploy(hw).then
    (function() {
        return deployer.deploy(hw1,hw.address);
    })};
```

7. 如下图所示那样执行 truffle.cmd 的 compile 命令。它可能会给出错误和警告。如果有任何错误和警告，应该在继续之前纠正它们：

```
C:\TruffleProject>truffle.cmd compile
Compiling .\contracts\First.sol...
Compiling .\contracts\Migrations.sol...
Compiling .\contracts\first.sol...
Compiling .\contracts\second.sol...

Compilation warnings encountered:

/C/TruffleProject/contracts/second.sol:9:5: Warning: No visibility specified. Defaulting to "public".
    function Second(address _first) {
    ^
Spanning multiple lines.

Writing artifacts to .\build\contracts
```

注意：在 Windows 上执行 truffle 命令时，如果它给出诸如未定义的模块之类错误信息时，你应该执行 truffle.cmd 命令，而不是抓住 truffle 命令不放。

8. 现在是时候来部署编译后的合约了。Truffle 提供了 migrate 命令，它

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

应该如下图所示被使用。值得注意的是，在运行 `migrate` 命令之前，前面网络配置里所配置需要链接的 Geth 或 ganache-cli 实例应该已经在运行了。如果使用 Geth，相应的挖矿进程也应该在运行。如果使用的是 `testrpc`，挖矿就没必要了：

```
C:\TruffleProject>truffle.cmd migrate
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x262e57282d269620a6642f6f5806ec72f0917d48979747d1cad9cb2106009a68
  Migrations: 0x65989fd1cdb5813460258a80c406ec25e00871a3
Saving successful migration to network...
  ... 0x7282c4bdde56076beaecd71785198ec2b93fd788a126ad40c06c37105d39402d
Saving artifacts...
Running migration: 2_Custom.js
  Deploying First...
  ... 0x3f507f2ecdc8842ca6f149d532401cfa7e325425960041dddb942e64381d7960
  First: 0xcf52edb0f5e9fd1509e5446b7c09889e0f3beb15
  Deploying Second...
  ... 0xc4bde155b2379d0ec8760e2da238dda96de6448291ae4410d98fef8e951df9a2
  Second: 0x7f3231d099966966230f1a0437d89f8824dc97db
Saving successful migration to network...
  ... 0x711f4234786b06542a48d374e0da98e6e348afd4d68d6247058c98595b51dd3b
Saving artifacts...
```

上面的屏幕截图显示了两个 migration 脚本都是根据它们的编号顺序来执行的。现在，合约已经被部署并可调用。合约地址伴随着交易散列也有了。可以使用合约的 ABI 定义和合约地址来创建合约实例了。

Truffle 还有很多的活动和命令，然而，为了保持章节内容的紧凑，我们将移步去理解使用 Truffle 进行智能合约单元测试。

9.4 使用 Truffle 进行测试

单元测试是对软件单元和组件进行独立验证的一种测试方式。单元测试有助于确保合约代码是根据功能和技术要求编写的。当每个最小组件在成功经过不同场景的测试后，就可以开展其他更重量级的测试，比如集成测试，去测试

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

多个组件。

如前所述，Truffle 会生成一个 test 目录，所有的测试文件都会放到该目录中。测试可以使用 Javascript 也可以使用 Solidity 编写。因为本书重点在 Solidity，这里会聚焦于使用 Solidity 来编写测试。

使用 Solidity 编写测试是通过创作合约文件来进行的。合约的名字应该以 Test 为前缀，每个合约函数应该以 test 为前缀。请注意这里的大小写区分。

下面的屏幕截图展示了在合约中编写测试的代码：

```
pragma solidity ^0.4.19;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/first.sol";

contract TestFirst {
  function testGetDoublePositiveUsingDeployedContract() {
    First meta = First(DeployedAddresses.First());

    Assert.equal(meta.GetDouble(10), 20, "Positive input gives double value");
  }

}
```

在这个 TestFirst 合约中有几个需要注意的地方。关键的 Truffle 库文件比如 Assert.sol 和 DeployAddresses.sol 被导入，以便于使用它们所提供的库函数。

这里为了演示方便只在合约中写了一个单元测试，实际上在同一个合约中可以写多个测试。

测试函数的第一行创建了一个对已部署合约 First 的引用，并随后调用了 GetDouble 函数。将此函数的返回值与 Assert.equal 函数的第二个参数进行比较，如果两者相同，则测试成功。否则就失败了。

非卖品，仅供非商业用途或交流学习使用

Assert.equal 函数有助于将实际返回值与预期返回值进行比较。

重要的是要理解，无论何时调用合约中的函数，它都是最终将被写入区块和账本的交易。实际上，测试合约功能也意味着你正在测试与该合约相关的交易。

下面的屏幕截图展示了使用 `test` 命令来执行测试。

```
C:\TruffleProject>truffle.cmd test
Using network 'development'.

Compiling .\contracts\first.sol...
Compiling .\test\TestFirst.sol...
Compiling truffle/Assert.sol...
Compiling truffle/DeployedAddresses.sol...

Compilation warnings encountered:

/C/TruffleProject/test/TestFirst.sol:10:3: Warning: No visibility specified. Defaulting to "public".
  function testGetDoublePositiveUsingDeployedContract() {
  ^
Spanning multiple lines.



  TestFirst
    √ testGetDoublePositiveUsingDeployedContract (110ms)


  1 passing (859ms)
```

9.5 本章小结

本章介绍了 Truffle 作为一种实用工具，用于简化创建、测试和部署 Solidity 合约的过程。Truffle 不是键入和执行每个步骤，而是提供了用于编译、部署和测试合约的简单命令。

接下来是本书的最后一章，将重点介绍与 Solidity 相关的故障排除活动与工具。调试是故障排除的一个重要方面，是任何合约开发人员和合约开发的重要技能。接下来会讨论 Remix 调试工具以及合约调试的其他相关机制。

非卖品，仅供非商业用途或交流学习使用

第 10 章 Chapter 10

合约调试

这是本书的最后一章。到目前为止，我们从概念上研究了 Solidity 和 Ethereum，开发并撰写了 Solidity 合约，并对其进行了测试。唯一没有讨论到的就是合约故障排查。故障排查在任何一门编程语言中都是一项重要技能和实践。它有助于发现问题并有效地解决这些问题。故障排查既是一门艺术，也是一门科学。开发人员应该通过实际经验来学习故障排查的技巧，并通过调试来探索幕后的细节。本章将重点介绍与 Solidity 合约相关的调试编码问题。

10.1 调试

在编写 Solidity 智能合约时，调试是一项重要实践。调试是指通过调整代码来查找问题、错误并移除它们。不管有没有足够的工具和实用程序支持，调试智能合约都不是一件容易的事情。通常，调试涉及逐步执行每行代码，查找临时、本地和全局变量的当前状态，并在执行合约时走查每条指令。

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

调试合约有下面几种方式：

- 使用 Remix 编辑器
- 事件
- BlockExplorer

10.1.1 Remix 编辑器

在先前的章节中我们使用 Remix 编辑器来编写智能合约。然而，我们还没有使用到 Remix 提供的调试工具。Remix 调试器通过帮助我们观察合约执行时的运行时行为来定位问题。它工作在 Solidity 及其生成的合约字节码中。可以暂停合约执行以检查合约代码、状态变量、局部变量和堆栈变量，并查看从合约代码生成的 EVM 指令。

下面的合约代码截图将用来演示使用 Remix 编辑器来进行调试工作：

```
pragma solidity ^0.4.0;

contract DebuggerSampleContract {

    int counter = 10;

    function LoopCounter(int _input) public view returns (int)  {
        int returnValue;

        for (; _input < counter; _input ++)
        {
            returnValue  += _input;
        }
        return returnValue;
    }
}
```

该合同有一个状态变量和函数。该函数循环提供的输入变量，直到达到 `counter` 的值并返回累计值给调用者。

如下面屏幕截图所示，在部署和执行 `LoopCounter` 函数时，我们有机会

严禁网络传播本PDF，违者责任自负！

通过点击 Debug 按钮来对该函数进行调试。

```
creation of DebuggerSampleContract pending...

[vm] from:0xca3...a733c, to:DebuggerSampleContract.(constructor), value:0 we
i, data:0x606...d0029, 0 logs, hash:0x719...ab533
```

Details　Debug

这将把焦点放在 Remix 中的 Debugger 选项卡上，在这里可以验证每一步代码执行所相关的本地变量、状态变量、内存信息、调用堆栈、堆栈信息、指令和调用数据的运行时信息。

以下两个屏幕截图显示了有关合约运行时的各种内部信息：

再看看第二张屏幕截图：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

下面这些从上面屏幕截图可以看到的指令，显示了函数执行时的字节码。

- SolidityLocals：该指令显示了输入参数、它们的数据类型和相应的值。
- SolidityState：该指令显示了状态变量、它们的类型及相应的当前值。
- StepDetail：对调试 gas 使用、消耗和剩余的 gas 量比较重要。
- CallStack：该指令显示了函数代码所需要的内部变量。
- Memory：该指令显示了函数内部的本地变量。
- CallData：该函数显示了客户端发送给合约的真正数据。头四个字节是函数标示，余下的 32 个字节是对应的输入参数。

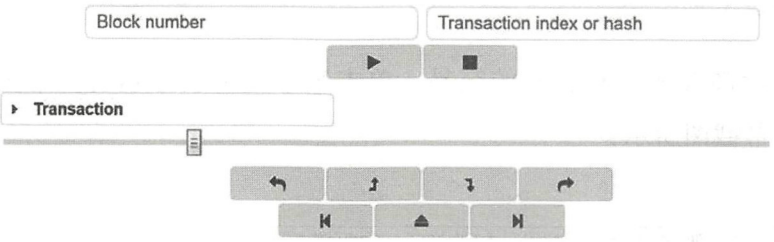
调试的一个重要方面是在每个感兴趣的代码行处都可以停止执行。断点有助于做到这一点。单击行号旁边的任何行都可以设置断点。再次单击将删除断点。在执行函数期间，当它到达此行时，执行暂停，可以从 Debugger 选项卡来验证值和执行。以下屏幕截图显示了断点：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！


```
3   pragma solidity ^0.4.0;
4
5   contract DebuggerSampleContract {
6
7       int counter = 10;
8
9       function LoopCounter(int _input) public view returns (int) {
10          int returnValue;
11
12          for (; _input < counter; _input ++)
13          {
14              returnValue  += _input;
15          }
16          return returnValue;
17      }
18  }
```

使用 Remix，还可以执行 Step over back、Step back、Step into、Step over forward，跳转到上一个断点，跳出，然后跳转到下一个断点。它还提供了使用特定区块号或交易散列值来查看相关信息的工具。甚至只是提供区块中的交易序列号而不是交易散列值也是有可能的，如下面屏幕截图所示：



10.1.2　使用事件

在第 8 章中我们已经看到如何使用事件。事件可以被捕获，并可以帮助提供有关当前执行的相关信息。合约应声明事件并且函数应该在适当的位置调用这些事件，为读取这些事件的人提供足够的上下文信息。

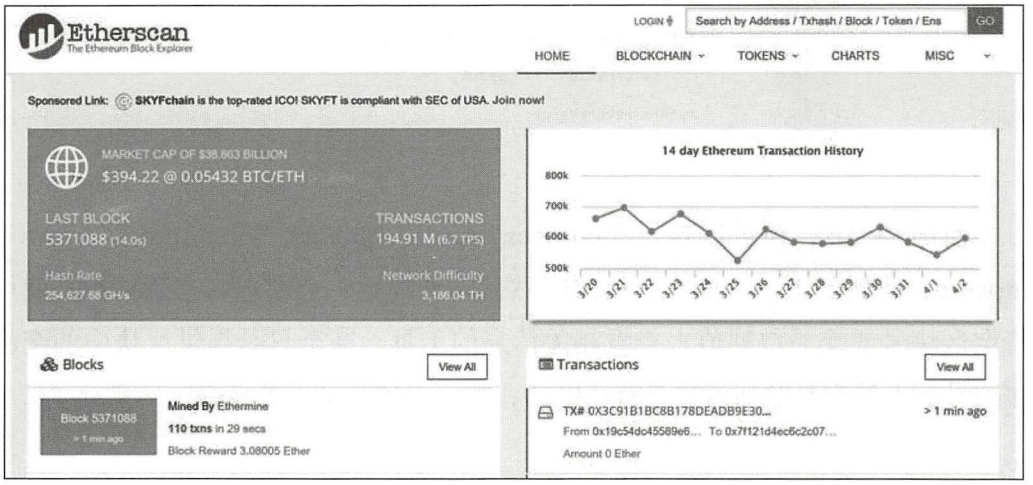
10.2　使用 Block Explorer

Block Explorer 就是一个以太坊浏览器。它提供有关其网络中当前区块和交

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

易的报告和信息。这是了解有关现有和过去更多数据信息的好地方。https://etherscan.io 就是个 Block Explorer，如下面屏幕截图所示：



它显示涉及账户和合同的交易。单击某个交易会显示有关它的详细信息，如以下屏幕截图所示：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

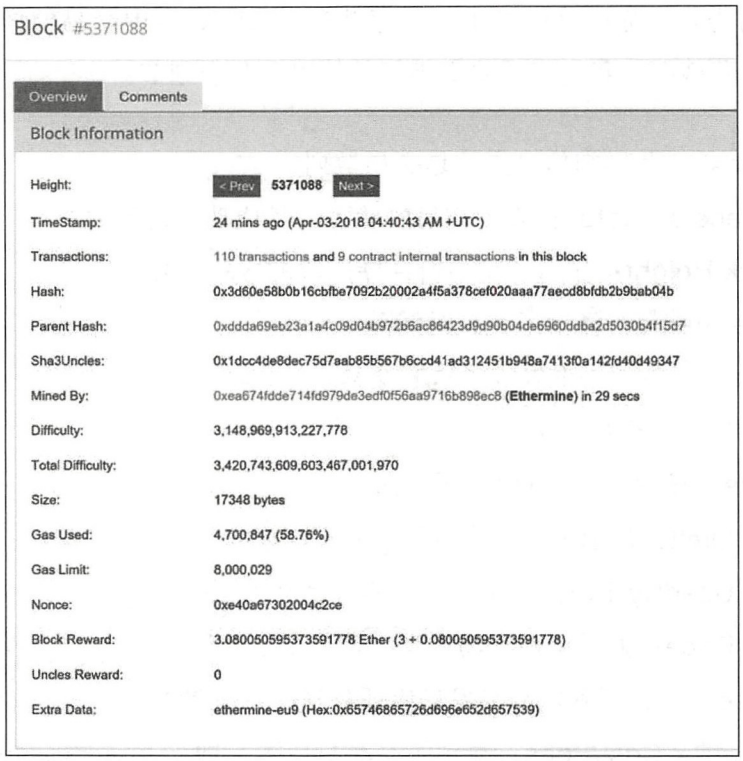
到目前为止，你应该已经了解了存储在以太坊账本中交易的详细信息。依照上面的屏幕截图，我们来看看交易相关的以下几个细节：

- TxHash：此详细信息指的是交易散列
- TxReceipt Status：表示交易的状态，是成功还是失败
- Block Height：显示了交易所存储区块的区块编号
- Time Stamp：显示了交易时间戳
- From：显示了是谁发送了这笔交易
- To：显示了交易的接收者
- Value：显示了发送的以太币数量
- Gas Limit：显示了用户指定的 gas 限定数量
- Gas UsedBy Txn：显示了交易实际消耗的 gas 数量
- Gas Price：显示了由交易发送者制定的 gas 价格
- nonce：是为了确定交易发送者所发送交易的数量
- Actual Tx Cost/Fees：显示了交易总成本，即 gasused*gasprice

单击区块可显示有关区块的信息以及属于该区块的交易列表。它显示了区块头中的所有详细信息，例如区块散列，父区块散列，矿工账户，难度级别，随机数等，如以下屏幕截图所示：

区块头里有一些有趣的属性，其中有一些会在这里提到。比如提供了在账本中区块编号的 Height、区块内的交易数（在本例中为 110），以及内部交易的数量（这些在合约之间被称为消息调用），当前区块头的散列值（Hash）、父块的散列值（Parent Hash）、叔块的根散列值、挖掘区块的 coinbase 或 etherbase 账户（Mined By）、当前区块的难度级别、到目前区块的累积难度、区块的大小、区块内所有交易使用的总 gas 量、区块的最大 gas 限量、用来证明工作证明的证据（Nonce）、以及挖矿奖励等。

非卖品，仅供非商业用途或交流学习使用


Block #5371088

Overview | Comments

Block Information

Height:	< Prev 5371088 Next >
TimeStamp:	24 mins ago (Apr-03-2018 04:40:43 AM +UTC)
Transactions:	110 transactions and 9 contract internal transactions in this block
Hash:	0x3d60e58b0b16cbfbe7092b20002a4f5a378cef020aaa77aecd8bfdb2b9bab04b
Parent Hash:	0xddda69eb23a1a4c09d04b972b6ac86423d9d90b04de6960ddba2d5030b4f15d7
Sha3Uncles:	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined By:	0xea674fdde714fd979de3edf0f56aa9716b898ec8 (Ethermine) in 29 secs
Difficulty:	3,148,969,913,227,778
Total Difficulty:	3,420,743,609,603,467,001,970
Size:	17348 bytes
Gas Used:	4,700,847 (58.76%)
Gas Limit:	8,000,029
Nonce:	0xe40a67302004c2ce
Block Reward:	3.080050595373591778 Ether (3 + 0.080050595373591778)
Uncles Reward:	0
Extra Data:	ethermine-eu9 (Hex:0x65746865726d696e652d657539)

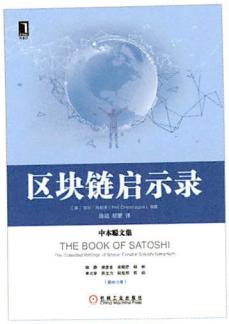
10.3 本章小结

Solidity 是一门不断发展改进的新编程语言。可以使用 Remix 编辑器来调试 Solidity 合约。Remix 通过在每一步都验证变量和代码执行，提供了一种编写和调试合约的便捷方式。它帮助我们在代码执行过程中方便地前进或后退。还可以用使断点来拦截代码的执行。也有其他调试合约的方式，比如使用 BlockExplorer 和 Solidity 事件。尽管 BlockExplorer 和事件提供的调试功能相当有限，但它们非常便于在生产环境中使用。

我希望阅读这本书对你来说是个蛮享受的过程，并由衷的相信你正在走向成为明星级 Solidity 开发者的路上。请保持关注并持续学习。

非卖品，仅供非商业用途或交流学习使用

推荐阅读



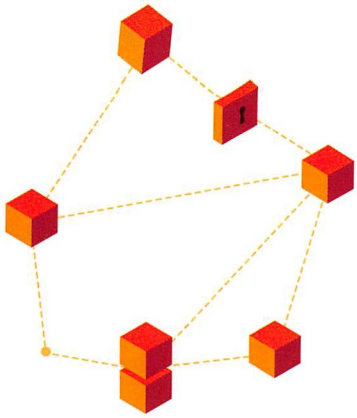
区块链启示录：中本聪文集
书号：978-7-111-60924-7

深入理解以太坊
书号：978-7-111-61492-0

区块链开发实战：
以太坊关键技术与案例分析
书号：978-7-111-59956-2

区块链项目开发指南
书号：978-7-111-58400-1

封面设计 · 陈潇然
非卖品，仅供非商业用途或交流学习使用



Solidity 是一种面向合约的语言，其语法受到 JavaScript 的很大影响，旨在帮助用户编写运行于以太坊虚拟机上的代码。本书将为你理解 Solidity 编程语言提供翔实参考，帮助你从头开构建以太坊和区块链的智能合约。

本书首先简要介绍区块链、以太坊及其相关的重要概念或组件；随后介绍如何安装所有必要工具来编写、测试和调试以太坊上的 Solidity 合约；之后探索 Solidity 源文件的布局及如何使用同的数据类型，介绍构建智能合约时使用的运算符、控制结构和数据结构；最后通过 Solidity 介面向对象编程中的函数调用、返回类型、函数修改器。通过本书，你将了解 Solidity 编写合约的质，掌握 Solidity 编程技能，能够在以太坊上编写、部署和测试智能合约。

通过阅读本书，你将：

- 学习 Solidity 和以太坊的基础知识和基本概念
- 深入探究 Solidity 语言及其独特性
- 详细了解如何使用 Solidity 编写智能合约
- 学习开发和部署智能合约的主要工具
- 使用异常处理和错误检查编写防御性代码
- 了解 Truffle 的基本知识和调试过程

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/区块链

ISBN 978-7-111-61600-

9 787111 61600

定价：59.00元